

Bitwise Operators	33
Other Operators	33
Working Out the Form Totals	36
Understanding Precedence and Associativity	37
Using Variable Handling Functions	39
Testing and Setting Variable Types	39
Testing Variable Status	40
Reinterpreting Variables	41
Making Decisions with Conditionals	41
if Statements	41
Code Blocks	42
else Statements	42
elseif Statements	43
switch Statements	44
Comparing the Different Conditionals	45
Repeating Actions Through Iteration	46
while Loops	47
for and foreach Loops	49
do...while Loops	50
Breaking Out of a Control Structure or Script	50
Employing Alternative Control Structure Syntax	51
Using declare	51
Next	52
2 Storing and Retrieving Data	53
Saving Data for Later	53
Storing and Retrieving Bob's Orders	54
Processing Files	55
Opening a File	55
Choosing File Modes	55
Using fopen() to Open a File	56
Opening Files Through FTP or HTTP	58
Addressing Problems Opening Files	58
Writing to a File	61
Parameters for fwrite()	62
File Formats	62
Closing a File	63

Reading from a File	65
Opening a File for Reading: <code>fopen()</code>	66
Knowing When to Stop: <code>feof()</code>	66
Reading a Line at a Time: <code>fgets()</code> , <code>fgetss()</code> , and <code>fgetcsw()</code>	67
Reading the Whole File: <code>readfile()</code> , <code>fpasssthru()</code> , <code>file()</code> , and <code>file_get_contents()</code>	68
Reading a Character: <code>fgetc()</code>	69
Reading an Arbitrary Length: <code>fread()</code>	69
Using Other File Functions	69
Checking Whether a File Is There: <code>file_exists()</code>	70
Determining How Big a File Is: <code>filesize()</code>	70
Deleting a File: <code>unlink()</code>	70
Navigating Inside a File: <code>rewind()</code> , <code>fseek()</code> , and <code>ftell()</code>	70
Locking Files	71
A Better Way: Databases	73
Problems with Using Flat Files	73
How RDBMSs Solve These Problems	74
Further Reading	74
Next	74

3 Using Arrays 75

What Is an Array?	75
Numerically Indexed Arrays	76
Initializing Numerically Indexed Arrays	76
Accessing Array Contents	77
Using Loops to Access the Array	78
Arrays with Different Indices	79
Initializing an Array	79
Accessing the Array Elements	79
Using Loops	79
Array Operators	81
Multidimensional Arrays	82
Sorting Arrays	85
Using <code>sort()</code>	85
Using <code>asort()</code> and <code>ksort()</code> to Sort Arrays	86
Sorting in Reverse	87

Sorting Multidimensional Arrays	87
Using the <code>array_multisort()</code> function	87
User-Defined Sorts	88
Reverse User Sorts	89
Reordering Arrays	90
Using <code>shuffle()</code>	90
Reversing an Array	92
Loading Arrays from Files	92
Performing Other Array Manipulations	96
Navigating Within an Array: <code>each()</code> , <code>current()</code> , <code>reset()</code> , <code>end()</code> , <code>next()</code> , <code>pos()</code> , and <code>prev()</code>	96
Applying Any Function to Each Element in an Array: <code>array_walk()</code>	97
Counting Elements in an Array: <code>count()</code> , <code>sizeof()</code> , and <code>array_count_values()</code>	98
Converting Arrays to Scalar Variables: <code>extract()</code>	99
Further Reading	100
Next	100
4 String Manipulation and Regular Expressions	101
Creating a Sample Application: Smart Form Mail	101
Formatting Strings	104
Trimming Strings: <code>chop()</code> , <code>ltrim()</code> , and <code>trim()</code>	104
Formatting Strings for Output	105
Joining and Splitting Strings with String Functions	112
Using <code>explode()</code> , <code>implode()</code> , and <code>join()</code>	112
Using <code>strtok()</code>	113
Using <code>substr()</code>	114
Comparing Strings	115
Performing String Ordering: <code>strcmp()</code> , <code>strcasecmp()</code> , and <code>strnatcmp()</code>	115
Testing String Length with <code>strlen()</code>	115
Matching and Replacing Substrings with String Functions	116
Finding Strings in Strings: <code>strstr()</code> , <code>strchr()</code> , <code>strrchr()</code> , and <code>stristr()</code>	116
Finding the Position of a Substring: <code>strpos()</code> and <code>strrpos()</code>	117
Replacing Substrings: <code>str_replace()</code> and <code>substr_replace()</code>	118

Introducing Regular Expressions	119
The Basics	120
Delimiters	120
Character Classes and Types	120
Repetition	122
Subexpressions	122
Counted Subexpressions	123
Anchoring to the Beginning or End of a String	123
Branching	123
Matching Literal Special Characters	123
Reviewing Meta Characters	124
Escape Sequences	125
Backreferences	126
Assertions	126
Putting It All Together for the Smart Form	127
Finding Substrings with Regular Expressions	128
Replacing Substrings with Regular Expressions	129
Splitting Strings with Regular Expressions	129
Further Reading	130
Next	130
5 Reusing Code and Writing Functions	131
The Advantages of Reusing Code	131
Cost	132
Reliability	132
Consistency	132
Using <code>require()</code> and <code>include()</code>	132
Using <code>require()</code> to Include Code	133
Using <code>require()</code> for Website Templates	134
Using <code>auto_prepend_file</code> and <code>auto_append_file</code>	139
Using Functions in PHP	140
Calling Functions	141
Calling an Undefined Function	142
Understanding Case and Function Names	143
Defining Your Own Functions	144
Examining Basic Function Structure	144
Naming Your Function	145
Using Parameters	146

Understanding Scope	148
Passing by Reference Versus Passing by Value	150
Using the <code>return</code> Keyword	152
Returning Values from Functions	153
Implementing Recursion	154
Implementing Anonymous Functions (or Closures)	155
Further Reading	157
Next	157

6 Object-Oriented PHP 159

Understanding Object-Oriented Concepts	160
Classes and Objects	160
Polymorphism	161
Inheritance	161
Creating Classes, Attributes, and Operations in PHP	162
Structure of a Class	162
Constructors	163
Destructors	163
Instantiating Classes	163
Using Class Attributes	164
Calling Class Operations	165
Controlling Access with <code>private</code> and <code>public</code>	166
Writing Accessor Functions	166
Implementing Inheritance in PHP	168
Controlling Visibility Through Inheritance with <code>private</code> and <code>protected</code>	169
Overriding	170
Preventing Inheritance and Overriding with <code>final</code>	172
Understanding Multiple Inheritance	172
Implementing Interfaces	173
Using Traits	174
Designing Classes	176
Writing the Code for Your Class	177
Understanding Advanced Object-Oriented Functionality in PHP	185
Using Per-Class Constants	185
Implementing Static Methods	185
Checking Class Type and Type Hinting	185

Late Static Bindings	186
Cloning Objects	187
Using Abstract Classes	188
Overloading Methods with <code>__call()</code>	188
Using <code>__autoload()</code>	189
Implementing Iterators and Iteration	190
Generators	192
Converting Your Classes to Strings	194
Using the Reflection API	194
Namespaces	195
Using Subnamespaces	197
Understanding the Global Namespace	197
Importing and Aliasing Namespaces	198
Next	198
7 Error and Exception Handling	199
Exception Handling Concepts	199
The <code>Exception</code> Class	201
User-Defined Exceptions	202
Exceptions in Bob's Auto Parts	204
Exceptions and PHP's Other Error Handling Mechanisms	208
Further Reading	208
Next	208

II: Using MySQL

8 Designing Your Web Database	209
Relational Database Concepts	210
Tables	210
Columns	211
Rows	211
Values	211
Keys	211
Schemas	212
Relationships	213
Designing Your Web Database	213
Think About the Real-World Objects You Are Modeling	213
Avoid Storing Redundant Data	214

Use Atomic Column Values	216
Choose Sensible Keys	217
Think About What You Want to Ask the Database	217
Avoid Designs with Many Empty Attributes	217
Summary of Table Types	218
Web Database Architecture	218
Further Reading	220
Next	220

9 Creating Your Web Database 221

Using the MySQL Monitor	222
Logging In to MySQL	223
Creating Databases and Users	224
Setting Up Users and Privileges	225
Introducing MySQL's Privilege System	225
Principle of Least Privilege	225
User Setup: The CREATE USER and GRANT Commands	225
Types and Levels of Privileges	227
The REVOKE Command	230
Examples Using GRANT and REVOKE	230
Setting Up a User for the Web	231
Using the Right Database	232
Creating Database Tables	232
Understanding What the Other Keywords Mean	234
Understanding the Column Types	235
Looking at the Database with SHOW and DESCRIBE	237
Creating Indexes	238
Understanding MySQL Identifiers	239
Choosing Column Data Types	240
Numeric Types	241
Date and Time Types	243
String Types	244
Further Reading	246
Next	246

10 Working with Your MySQL Database 247

What Is SQL?	247
Inserting Data into the Database	248

Retrieving Data from the Database	250
Retrieving Data with Specific Criteria	251
Retrieving Data from Multiple Tables	253
Retrieving Data in a Particular Order	259
Grouping and Aggregating Data	259
Choosing Which Rows to Return	261
Using Subqueries	262
Updating Records in the Database	265
Altering Tables After Creation	265
Deleting Records from the Database	268
Dropping Tables	268
Dropping a Whole Database	268
Further Reading	269
Next	269
11 Accessing Your MySQL Database from the Web with PHP	271
How Web Database Architectures Work	272
Querying a Database from the Web	275
Checking and Filtering Input Data	276
Setting Up a Connection	277
Choosing a Database to Use	278
Querying the Database	278
Using Prepared Statements	279
Retrieving the Query Results	280
Disconnecting from the Database	281
Putting New Information in the Database	282
Using Other PHP-Database Interfaces	286
Using a Generic Database Interface: PDO	286
Further Reading	289
Next	289
12 Advanced MySQL Administration	291
Understanding the Privilege System in Detail	291
The user Table	293
The db Table	295
The tables_priv, columns_priv, and procs_priv Tables	296
Access Control: How MySQL Uses the Grant Tables	298
Updating Privileges: When Do Changes Take Effect?	299

Making Your MySQL Database Secure	299
MySQL from the Operating System's Point of View	299
Passwords	300
User Privileges	300
Web Issues	301
Getting More Information About Databases	301
Getting Information with SHOW	302
Getting Information About Columns with DESCRIBE	304
Understanding How Queries Work with EXPLAIN	304
Optimizing Your Database	309
Design Optimization	309
Permissions	309
Table Optimization	310
Using Indexes	310
Using Default Values	310
Other Tips	310
Backing Up Your MySQL Database	310
Restoring Your MySQL Database	311
Implementing Replication	311
Setting Up the Master	312
Performing the Initial Data Transfer	313
Setting Up the Slave or Slaves	313
Further Reading	314
Next	314

13 Advanced MySQL Programming 315

The LOAD DATA INFILE Statement	315
Storage Engines	316
Transactions	317
Understanding Transaction Definitions	317
Using Transactions with InnoDB	318
Foreign Keys	319
Stored Procedures	320
Basic Example	320
Local Variables	323
Cursors and Control Structures	323

Triggers	327
Further Reading	329
Next	329

III: Web Application Security

14	Web Application Security Risks	331
	Identifying the Threats We Face	331
	Access to Sensitive Data	331
	Modification of Data	334
	Loss or Destruction of Data	334
	Denial of Service	335
	Malicious Code Injection	337
	Compromised Server	338
	Repudiation	338
	Understanding Who We're Dealing With	339
	Attackers and Crackers	339
	Unwitting Users of Infected Machines	339
	Disgruntled Employees	339
	Hardware Thieves	340
	Ourselves	340
	Next	340
15	Building a Secure Web Application	341
	Strategies for Dealing with Security	341
	Start with the Right Mindset	342
	Balancing Security and Usability	342
	Monitoring Security	342
	Our Basic Approach	343
	Securing Your Code	343
	Filtering User Input	343
	Escaping Output	348
	Code Organization	350
	What Goes in Your Code	351
	File System Considerations	352
	Code Stability and Bugs	352
	Executing Commands	353

Securing Your Web Server and PHP	354
Keep Software Up-to-Date	354
Browse the <code>php.ini</code> file	355
Web Server Configuration	356
Shared Hosting of Web Applications	356
Database Server Security	357
Users and the Permissions System	358
Sending Data to the Server	358
Connecting to the Server	359
Running the Server	359
Protecting the Network	360
Firewalls	360
Use a DMZ	360
Prepare for DoS and DDoS Attacks	361
Computer and Operating System Security	361
Keep the Operating System Up to Date	361
Run Only What Is Necessary	362
Physically Secure the Server	362
Disaster Planning	362
Next	364

16 Implementing Authentication Methods with PHP 365

Identifying Visitors	365
Implementing Access Control	366
Storing Passwords	369
Securing Passwords	369
Protecting Multiple Pages	371
Using Basic Authentication	372
Using Basic Authentication in PHP	372
Using Basic Authentication with Apache's <code>.htaccess</code> Files	374
Creating Your Own Custom Authentication	377
Further Reading	377
Next	377

IV: Advanced PHP Techniques

17 Interacting with the File System and the Server 379

Uploading Files	379
HTML for File Upload	381

Writing the PHP to Deal with the File	382
Session Upload Progress	387
Avoiding Common Upload Problems	389
Using Directory Functions	390
Reading from Directories	390
Getting Information About the Current Directory	394
Creating and Deleting Directories	394
Interacting with the File System	395
Getting File Information	395
Changing File Properties	397
Creating, Deleting, and Moving Files	398
Using Program Execution Functions	398
Interacting with the Environment: <code>getenv()</code> and <code>putenv()</code>	401
Further Reading	402
Next	402

18 Using Network and Protocol Functions 403

Examining Available Protocols	403
Sending and Reading Email	404
Using Data from Other Websites	404
Using Network Lookup Functions	408
Backing Up or Mirroring a File	412
Using FTP to Back Up or Mirror a File	412
Uploading Files	420
Avoiding Timeouts	420
Using Other FTP Functions	420
Further Reading	421
Next	421

19 Managing the Date and Time 423

Getting the Date and Time from PHP	423
Understanding Timezones	423
Using the <code>date()</code> Function	424
Dealing with Unix Timestamps	426
Using the <code>getdate()</code> Function	427
Validating Dates with <code>checkdate()</code>	428
Formatting Timestamps	429
Converting Between PHP and MySQL Date Formats	431

- Calculating Dates in PHP 433
- Calculating Dates in MySQL 434
- Using Microseconds 435
- Using the Calendar Functions 436
- Further Reading 436
- Next 436

20 Internationalization and Localization 437

- Localization Is More than Translation 437
- Understanding Character Sets 438
 - Security Implications of Character Sets 439
 - Using Multibyte String Functions in PHP 440
- Creating a Basic Localizable Page Structure 440
- Using `gettext()` in an Internationalized Application 444
 - Configuring Your System to Use `gettext()` 444
 - Creating Translation Files 445
 - Implementing Localized Content in PHP Using `gettext()` 447
- Further Reading 448
- Next 448

21 Generating Images 449

- Setting Up Image Support in PHP 449
- Understanding Image Formats 450
 - JPEG 450
 - PNG 450
 - GIF 451
- Creating Images 451
 - Creating a Canvas Image 452
 - Drawing or Printing Text on the Image 453
 - Outputting the Final Graphic 455
 - Cleaning Up 455
- Using Automatically Generated Images in Other Pages 456
- Using Text and Fonts to Create Images 457
 - Setting Up the Base Canvas 460
 - Fitting the Text onto the Button 461
 - Positioning the Text 464
 - Writing the Text onto the Button 464
 - Finishing Up 465

Drawing Figures and Graphing Data 465
 Using Other Image Functions 474
 Next 474

22 Using Session Control in PHP 475

What Is Session Control? 475
 Understanding Basic Session Functionality 476
 What Is a Cookie? 476
 Setting Cookies from PHP 476
 Using Cookies with Sessions 477
 Storing the Session ID 477
 Implementing Simple Sessions 478
 Starting a Session 478
 Registering Session Variables 478
 Using Session Variables 479
 Unsetting Variables and Destroying the Session 479
 Creating a Simple Session Example 480
 Configuring Session Control 482
 Implementing Authentication with Session Control 483
 Next 491

23 Integrating JavaScript and PHP 493

Understanding AJAX 493
 A Brief Introduction to jQuery 494
 Using jQuery in Web Applications 494
 Using jQuery and AJAX with PHP 504
 The AJAX-Enabled Chat Script/Server 504
 The jQuery AJAX Methods 507
 The Chat Client/jQuery Application 510
 Further Reading 517
 Next 517

24 Other Useful Features 519

Evaluating Strings: `eval()` 519
 Terminating Execution: `die()` and `exit()` 520
 Serializing Variables and Objects 521
 Getting Information About the PHP Environment 522
 Finding Out What Extensions Are Loaded 522

Identifying the Script Owner	523
Finding Out When the Script Was Modified	523
Temporarily Altering the Runtime Environment	524
Highlighting Source Code	525
Using PHP on the Command Line	526
Next	527

V: Building Practical PHP and MySQL Projects

25 Using PHP and MySQL for Large Projects 529

Applying Software Engineering to Web Development	530
Planning and Running a Web Application Project	530
Reusing Code	531
Writing Maintainable Code	532
Coding Standards	532
Breaking Up Code	535
Using a Standard Directory Structure	536
Documenting and Sharing In-House Functions	536
Implementing Version Control	536
Choosing a Development Environment	537
Documenting Your Projects	538
Prototyping	538
Separating Logic and Content	539
Optimizing Code	540
Using Simple Optimizations	540
Testing	541
Further Reading	542
Next	542

26 Debugging and Logging 543

Programming Errors	543
Syntax Errors	543
Runtime Errors	544
Logic Errors	549
Variable Debugging Aid	551
Error Reporting Levels	553
Altering the Error Reporting Settings	554
Triggering Your Own Errors	556

- Logging Errors Gracefully 557
- Logging Errors to a Log File 560
- Next 560

27 Building User Authentication and Personalization 561

- Solution Components 561
 - User Identification and Personalization 562
 - Storing Bookmarks 563
 - Recommending Bookmarks 563
- Solution Overview 563
- Implementing the Database 565
- Implementing the Basic Site 566
- Implementing User Authentication 569
 - Registering Users 569
 - Logging In 575
 - Logging Out 579
 - Changing Passwords 580
 - Resetting Forgotten Passwords 582
- Implementing Bookmark Storage and Retrieval 587
 - Adding Bookmarks 588
 - Displaying Bookmarks 590
 - Deleting Bookmarks 591
- Implementing Recommendations 594
- Considering Possible Extensions 598

28 Building a Web-Based Email Service with Laravel Part I Web Edition

29 Building a Web-Based Email Service with Laravel Part II Web Edition

30 Social Media Integration Sharing and Authentication Web Edition

31 Building a Shopping Cart Web Edition

VI: Appendix

A Installing Apache, PHP, and MySQL 599

- Installing Apache, PHP, and MySQL Under UNIX 600
 - Binary Installation 600
 - Source Installation 601
 - Basic Apache Configuration Modifications 608

Is PHP Support Working? 610

Is SSL Working? 610

Installing Apache, PHP, and MySQL for Windows and Mac OS X
Using All-in-One Installation Packages 612

Installing PEAR 613

Installing PHP with Other Web Servers 614

Index 615

Lead Authors

Laura Thomson is Director of Engineering at Mozilla Corporation. She was formerly a principal at both OmniTI and Tangled Web Design, and she has worked for RMIT University and the Boston Consulting Group. She holds a Bachelor of Applied Science (Computer Science) degree and a Bachelor of Engineering (Computer Systems Engineering) degree with honors. In her spare time she enjoys riding horses, arguing about free and open source software, and sleeping.

Luke Welling is a software engineer and regularly speaks on open source and web development topics at conferences such as OSCON, ZendCon, MySQLUC, PHPCon, OSDC, and LinuxTag. He has worked for OmniTI, for the web analytics company Hitwise.com, at the database vendor MySQL AB, and as an independent consultant at Tangled Web Design. He has taught computer science at RMIT University in Melbourne, Australia, and holds a Bachelor of Applied Science (Computer Science) degree. In his spare time, he attempts to perfect his insomnia.

Contributing Authors

Julie C. Meloni is a software development manager and technical consultant living in Washington, D.C. She has written several books and articles on web-based programming languages and database topics, including the bestselling *Sams Teach Yourself PHP, MySQL and Apache All in One*.

John Coggeshall is the owner of Internet Technology Solutions, LLC—an Internet and PHP consultancy serving customers worldwide, as well as the owner of CoogletNet, a subscription based WiFi network. As former senior member of Zend Technologies' Global Services team, he got started with PHP in 1997 and is the author of four published books and over 100 articles on PHP technologies.

Jennifer Kyrnin is an author and web designer who has been working on the Internet since 1995. Her other books include *Sams Teach Yourself Bootstrap in 24 Hours*, *Sams Teach Yourself Responsive Web Design in 24 Hours*, and *Sams Teach Yourself HTML5 Mobile Application Development in 24 Hours*.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write directly to let us know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and phone or email address.

Email: feedback@developers-library.info

Mail: Reader Feedback
 Addison-Wesley Developer's Library
 800 East 96th Street
 Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Accessing the Free Web Edition

Your purchase of this book in any format, print or electronic, includes access to the corresponding Web Edition, which provides several special features to help you learn:

- The complete text of the book online
- Interactive quizzes and exercises to test your understanding of the material
- Bonus chapters not included in the print or e-book editions
- Updates and corrections as they become available

The Web Edition can be viewed on all types of computers and mobile devices with any modern web browser that supports HTML5.

To get access to the Web Edition of *PHP and MySQL Web Development, Fifth Edition*, all you need to do is register this book:

1. Go to www.informit.com/register
2. Sign in or create a new account
3. Enter ISBN: 9780321833891
4. Answer the questions as proof of purchase

The Web Edition will appear under the Digital Purchases tab on your Account page. Click the Launch link to access the product.

This page intentionally left blank

Introduction

Welcome to *PHP and MySQL Web Development*. Within its pages, you will find distilled knowledge from our experiences using PHP and MySQL, two of the most important and widely used web development tools around.

Key topics covered in this introduction include

- Why you should read this book
- What you will be able to achieve using this book
- What PHP and MySQL are and why they're great
- What's changed in the latest versions of PHP and MySQL
- How this book is organized

Let's get started.

Note

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Why You Should Read This Book

This book will teach you how to create interactive web applications from the simplest order form through to complex, secure web applications. What's more, you'll learn how to do it using open-source technologies.

This book is aimed at readers who already know at least the basics of HTML and have done some programming in a modern programming language before but have not necessarily programmed for the web or used a relational database. If you are a beginning programmer, you should still find this book useful, but digesting it might take a little longer. We've tried not to leave out any basic concepts, but we do cover them at speed. The typical readers of this book want to master PHP and MySQL for the purpose of building a large or commercial website. You might already be working in another web development language; if so, this book should get you up to speed quickly.

We wrote the first edition of this book because we were tired of finding PHP books that were basically function references. These books are useful, but they don't help when your boss or client has said, "Go build me a shopping cart." In this book, we have done our best to make every example useful. You can use many of the code samples directly in your website, and you can use many others with only minor modifications.

What You Will Learn from This Book

Reading this book will enable you to build real-world, dynamic web applications. If you've built websites using plain HTML, you realize the limitations of this approach. Static content from a pure HTML website is just that—static. It stays the same unless you physically update it. Your users can't interact with the site in any meaningful fashion.

Using a language such as PHP and a database such as MySQL allows you to make your sites dynamic: to have them be customizable and contain real-time information.

We have deliberately focused this book on real-world applications, even in the introductory chapters. We begin by looking at simple systems and work our way through the various parts of PHP and MySQL.

We then discuss aspects of security and authentication as they relate to building a real-world website and show you how to implement these aspects in PHP and MySQL. We also introduce you to integrating front-end and back-end technologies by discussing JavaScript and the role it can play in your application development.

In the final part of this book, we describe how to approach real-world projects and take you through the design, planning, and building of the following projects:

- User authentication and personalization
- Web-based email
- Social media integration

You should be able to use any of these projects as is, or you can modify them to suit your needs. We chose them because we believe they represent some of the most common web applications built by programmers. If your needs are different, this book should help you along the way to achieving your goals.

What Is PHP?

PHP is a server-side scripting language designed specifically for the web. Within an HTML page, you can embed PHP code that will be executed each time the page is visited. Your PHP code is interpreted at the web server and generates HTML or other output that the visitor will see.

PHP was conceived in 1994 and was originally the work of one man, Rasmus Lerdorf. It was adopted by other talented people and has gone through several major rewrites to bring us the

broad, mature product we see today. According to Google's Greg Michillie in May 2013, PHP ran more than three quarters of the world's websites, and that number had grown to over 82% by July 2016.

PHP is an open-source project, which means you have access to the source code and have the freedom to use, alter, and redistribute it.

PHP originally stood for *Personal Home Page* but was changed in line with the GNU recursive naming convention (GNU = Gnu's Not Unix) and now stands for *PHP Hypertext Preprocessor*.

The current major version of PHP is 7. This version saw a complete rewrite of the underlying Zend engine and some major improvements to the language. All of the code in this book has been tested and validated against the most recent release of PHP 7 at the time of writing, as well as the latest version in the PHP 5.6 family of releases, which is still officially supported.

The home page for PHP is available at <http://www.php.net>.

The home page for Zend Technologies is <http://www.zend.com>.

What Is MySQL?

MySQL (pronounced *My-Ess-Que-El*) is a very fast, robust, *relational database management system* (RDBMS). A database enables you to efficiently store, search, sort, and retrieve data. The MySQL server controls access to your data to ensure that multiple users can work with it concurrently, to provide fast access to it, and to ensure that only authorized users can obtain access. Hence, MySQL is a multiuser, multithreaded server. It uses *Structured Query Language* (SQL), the standard database query language. MySQL has been publicly available since 1996 but has a development history going back to 1979. It is the world's most popular open-source database and has won the Linux Journal Readers' Choice Award on a number of occasions.

MySQL is available under a dual licensing scheme. You can use it under an open-source license (the GPL) free as long as you are willing to meet the terms of that license. If you want to distribute a non-GPL application including MySQL, you can buy a commercial license instead.

Why Use PHP and MySQL?

When setting out to build a website, you could use many different products.

You need to choose the following:

- Where to run your web servers: the cloud, virtual private servers, or actual hardware
- An operating system
- Web server software
- A database management system or other datastore
- A programming or scripting language

You may end up with a hybrid architecture with multiple datastores. Some of these choices are dependent on the others. For example, not all operating systems run on all hardware, not all web servers support all programming languages, and so on.

In this book, we do not pay much attention to hardware, operating systems, or web server software. We don't need to. One of the best features of both PHP and MySQL is that they work with any major operating system and many of the minor ones.

The majority of PHP code can be written to be portable between operating systems and web servers. There are some PHP functions that specifically relate to the filesystem that are operating system dependent, but these are clearly marked as such in the manual and in this book.

Whatever hardware, operating system, and web server you choose, we believe you should seriously consider using PHP and MySQL.

Some of PHP's Strengths

Some of PHP's main competitors are Python, Ruby (on Rails or otherwise), Node.js, Perl, Microsoft .NET, and Java.

In comparison to these products, PHP has many strengths, including the following:

- Performance
- Scalability
- Interfaces to many different database systems
- Built-in libraries for many common web tasks
- Low cost
- Ease of learning and use
- Strong object-oriented support
- Portability
- Flexibility of development approach
- Availability of source code
- Availability of support and documentation

A more detailed discussion of these strengths follows.

Performance

PHP is very fast. Using a single inexpensive server, you can serve millions of hits per day. It scales down to the smallest email form and up to sites such as Facebook and Etsy.

Scalability

PHP has what Rasmus Lerdorf frequently refers to as a “shared-nothing” architecture. This means that you can effectively and cheaply implement horizontal scaling with large numbers of commodity servers.

Database Integration

PHP has native connections available to many database systems. In addition to MySQL, you can directly connect to PostgreSQL, Oracle, MongoDB, and MSSQL, among others. PHP 5 and PHP 7 also have a built-in SQL interface to flat files, called SQLite.

Using the *Open Database Connectivity* (ODBC) standard, you can connect to any database that provides an ODBC driver. This includes Microsoft products and many others.

In addition to native libraries, PHP comes with a database access abstraction layer called *PHP Database Objects* (PDOs), which allows consistent access and promotes secure coding practices.

Built-in Libraries

Because PHP was designed for use on the Web, it has many built-in functions for performing many useful web-related tasks. You can generate images on the fly, connect to web services and other network services, parse XML, send email, work with cookies, and generate PDF documents, all with just a few lines of code.

Cost

PHP is free. You can download the latest version at any time from <http://www.php.net> for no charge.

Ease of Learning PHP

The syntax of PHP is based on other programming languages, primarily C and Perl. If you already know C or Perl, or a C-like language such as C++ or Java, you will be productive using PHP almost immediately.

Object-Oriented Support

PHP version 5 had well-designed object-oriented features, which continued to be refined and improved in PHP version 7. If you learned to program in Java or C++, you will find the features (and generally the syntax) that you expect, such as inheritance, private and protected attributes and methods, abstract classes and methods, interfaces, constructors, and destructors. You will even find some less common features such as iterators and traits.

Portability

PHP is available for many different operating systems. You can write PHP code on free UNIX-like operating systems such as Linux and FreeBSD, commercial UNIX versions, OS X, or on different versions of Microsoft Windows.

Well-written code will usually work without modification on a different system running PHP.

Flexibility of Development Approach

PHP allows you to implement simple tasks simply, and equally easily adapts to implementing large applications using a framework based on design patterns such as Model-View-Controller (MVC).

Source Code

You have access to PHP's source code. With PHP, unlike commercial, closed-source products, if you want to modify something or add to the language, you are free to do so.

You do not need to wait for the manufacturer to release patches. You also don't need to worry about the manufacturer going out of business or deciding to stop supporting a product.

Availability of Support and Documentation

Zend Technologies (<http://www.zend.com>), the company behind the engine that powers PHP, funds its PHP development by offering support and related software on a commercial basis.

The PHP documentation and community are mature and rich resources with a wealth of information to share.

Key Features of PHP 7

In December 2015, the long-awaited PHP 7 release was made available to the public. As mentioned in this introduction, the book covers both PHP 5.6 and PHP 7, which might lead you to ask “what happened to PHP 6?” The short answer is: there is no PHP 6 and never was for the general public. There was a development effort around a codebase that was referred to as “PHP 6” but it never came to fruition; there were many ambitious plans and subsequent complications that made it difficult for the team to continue to pursue. PHP 7 is *not* PHP 6 and doesn't include the features and code from that development effort; PHP 7 is its own release with its own focus—specifically a focus on performance.

Under the hood, PHP 7 includes a refactor of the Zend Engine that powers it, which resulted in a significant performance boost to many web applications—sometimes upwards of 100%! While increased performance and decreased memory use were key to the release of PHP 7, so was backward-compatibility. In fact, relatively few backward-incompatible language changes were introduced. These are discussed contextually throughout this book so that the chapters

remain usable with PHP 5.6 or PHP 7, as widespread adoption of PHP 7 has not yet occurred by commercial web-hosting providers.

Some of MySQL's Strengths

MySQL's main competitors in the relational database space are PostgreSQL, Microsoft SQL Server, and Oracle. There is also a growing trend in the web application world toward use of NoSQL/non-relational databases such as MongoDB. Let's take a look at why MySQL is still a good choice in many cases.

MySQL has many strengths, including the following:

- High performance
- Low cost
- Ease of configuration and learning
- Portability
- Availability of source code
- Availability of support

A more detailed discussion of these strengths follows.

Performance

MySQL is undeniably fast. You can see the developers' benchmark page at <http://www.mysql.com/why-mysql/benchmarks/>.

Low Cost

MySQL is available at no cost under an open-source license or at low cost under a commercial license. You need a license if you want to redistribute MySQL as part of an application and do not want to license your application under an open-source license. If you do not intend to distribute your application—typical for most web applications—or are working on free or open-source software, you do not need to buy a license.

Ease of Use

Most modern databases use SQL. If you have used another RDBMS, you should have no trouble adapting to this one. MySQL is also easier to set up and tune than many similar products.

Portability

MySQL can be used on many different UNIX systems as well as under Microsoft Windows.

Source Code

As with PHP, you can obtain and modify the source code for MySQL. This point is not important to most users most of the time, but it provides you with excellent peace of mind, ensuring future continuity and giving you options in an emergency.

In fact, there are now several forks and drop-in replacements for MySQL that you may consider using, including MariaDB, written by the original authors of MySQL, including Michael ‘Monty’ Widenius (<https://mariadb.org>).

Availability of Support

Not all open-source products have a parent company offering support, training, consulting, and certification, but you can get all of these benefits from Oracle (who acquired MySQL with their acquisition of Sun Microsystems, who had previously acquired the founding company, MySQL AB).

What Is New in MySQL (5.x)?

At the time of writing, the current version of MySQL was 5.7.

Features added to MySQL in the last few releases include

- A wide range of security improvements
- FULLTEXT support for InnoDB tables
- A NoSQL-style API for InnoDB
- Partitioning support
- Improvements to replication, including row-based replication and GTIDs
- Thread pooling
- Pluggable authentication
- Multicore scalability
- Better diagnostic tools
- InnoDB as the default engine
- IPv6 support
- Plugin API
- Event scheduling
- Automated upgrades

Other changes include more ANSI standard compliance and performance improvements.

If you are still using an early 4.x version or a 3.x version of the MySQL server, you should know that the following features were added to various versions from 4.0:

- Views
- Stored procedures
- Triggers and cursors
- Subquery support
- GIS types for storing geographical data
- Improved support for internationalization
- The transaction-safe storage engine InnoDB included as standard
- The MySQL query cache, which greatly improves the speed of repetitive queries as often run by web applications

How Is This Book Organized?

This book is divided into five main parts:

Part I, “Using PHP,” provides an overview of the main parts of the PHP language with examples. Each example is a real-world example used in building an e-commerce site rather than “toy” code. We kick off this section with Chapter 1, “PHP Crash Course.” If you’ve already used PHP, you can whiz through this chapter. If you are new to PHP or new to programming, you might want to spend a little more time on it.

Part II, “Using MySQL,” discusses the concepts and design involved in using relational database systems such as MySQL, using SQL, connecting your MySQL database to the world with PHP, and advanced MySQL topics, such as security and optimization.

Part III, “Web Application Security,” covers some of the general issues involved in developing a web application using any language. We then discuss how you can use PHP and MySQL to authenticate your users and securely gather, transmit, and store data.

Part IV, “Advanced PHP Techniques,” offers detailed coverage of some of the major built-in functions in PHP. We have selected groups of functions that are likely to be useful when building a web application. You will learn about interaction with the server, interaction with the network, image generation, date and time manipulation, and session handling.

Part V, “Building Practical PHP and MySQL Projects,” is our favorite section. It deals with practical real-world issues such as managing large projects and debugging, and provides sample projects that demonstrate the power and versatility of PHP and MySQL.

Accessing the Free Web Edition

Your purchase of this book in any format includes access to the corresponding Web Edition, which provides several special features to help you learn:

- The complete text of the book online
- Interactive quizzes and exercises to test your understanding of the material
- Bonus chapters not included in the print or e-book editions
- Updates and corrections as they become available

The Web Edition can be viewed on all types of computers and mobile devices with any modern web browser that supports HTML5.

To get access to the Web Edition of *PHP and MySQL Web Development, Fifth Edition* all you need to do is register this book:

1. Go to www.informit.com/register
2. Sign in or create a new account
3. Enter ISBN: 9780321833891
4. Answer the questions as proof of purchase

The Web Edition will appear under the Digital Purchases tab on your Account page. Click the Launch link to access the product.

Finally

We hope you enjoy this book and enjoy learning about PHP and MySQL as much as we did when we first began using these products. They are really a pleasure to use. Soon, you'll be able to join the many thousands of web developers who use these robust, powerful tools to easily build dynamic, real-time web applications.

PHP Crash Course

This chapter gives you a quick overview of PHP syntax and language constructs. If you are already a PHP programmer, it might fill some gaps in your knowledge. If you have a background using C, Perl, Python, or another programming language, it will help you get up to speed quickly.

In this book, you'll learn how to use PHP by working through lots of real-world examples taken from our experiences building real websites. Often, programming textbooks teach basic syntax with very simple examples. We have chosen not to do that. We recognize that what you do is get something up and running, and understand how the language is used, instead of plowing through yet another syntax and function reference that's no better than the online manual.

Try the examples. Type them in or download them from the website, change them, break them, and learn how to fix them again.

This chapter begins with the example of an online product order form to show how variables, operators, and expressions are used in PHP. It also covers variable types and operator precedence. You will learn how to access form variables and manipulate them by working out the total and tax on a customer order.

You will then develop the online order form example by using a PHP script to validate the input data. You'll examine the concept of Boolean values and look at examples using `if`, `else`, the `?:` operator, and the `switch` statement. Finally, you'll explore looping by writing some PHP to generate repetitive HTML tables.

Key topics you learn in this chapter include

- Embedding PHP in HTML
- Adding dynamic content
- Accessing form variables
- Understanding identifiers

- Creating user-declared variables
- Examining variable types
- Assigning values to variables
- Declaring and using constants
- Understanding variable scope
- Understanding operators and precedence
- Evaluating expressions
- Using variable functions
- Making decisions with `if`, `else`, and `switch`
- Taking advantage of iteration using `while`, `do`, and `for` loops

Before You Begin: Accessing PHP

To work through the examples in this chapter and the rest of the book, you need access to a web server with PHP installed. To gain the most from the examples and case studies, you should run them and try changing them. To do this, you need a testbed where you can experiment.

If PHP is not installed on your machine, you need to begin by installing it or having your system administrator install it for you. You can find instructions for doing so in Appendix A, “Installing Apache, PHP, and MySQL.”

Creating a Sample Application: Bob’s Auto Parts

One of the most common applications of any server-side scripting language is processing HTML forms. You’ll start learning PHP by implementing an order form for Bob’s Auto Parts, a fictional spare parts company. You can find all the code for the examples used in this chapter in the directory called `chapter01` on the CD-ROM.

Creating the Order Form

Bob’s HTML programmer has set up an order form for the parts that Bob sells. This relatively simple order form, shown in Figure 1.1, is similar to many you have probably seen while surfing. Bob would like to be able to know what his customers ordered, work out the total prices of their orders, and determine how much sales tax is payable on the orders.

Item	Quantity
Tires	<input type="text"/>
Oil	<input type="text"/>
Spark Plugs	<input type="text"/>

Figure 1.1 Bob's initial order form records only products and quantities

Part of the HTML for this form is shown in Listing 1.1.

Listing 1.1 **orderform.html**— HTML for Bob's Basic Order Form

```
<form action="processorder.php" method="post">
<table style="border: 0px;">
<tr style="background: #cccccc;">
  <td style="width: 150px; text-align: center;">Item</td>
  <td style="width: 15px; text-align: center;">Quantity</td>
</tr>
<tr>
  <td>Tires</td>
  <td><input type="text" name="tireqty" size="3"
    maxlength="3" /></td>
</tr>
<tr>
  <td>Oil</td>
  <td><input type="text" name="oilqty" size="3"
    maxlength="3" /></td>
</tr>
<tr>
  <td>Spark Plugs</td>
  <td><input type="text" name="sparkqty" size="3"
    maxlength="3" /></td>
</tr>
<tr>
```

```

        <td colspan="2" style="text-align: center;"><input type="submit" value="Submit
Order" /></td>
    </tr>
</table>
</form>

```

Notice that the form's action is set to the name of the PHP script that will process the customer's order. (You'll write this script next.) In general, the value of the `action` attribute is the URL that will be loaded when the user clicks the Submit button. The data the user has typed in the form will be sent to this URL via the HTTP method specified in the `method` attribute, either `get` (appended to the end of the URL) or `post` (sent as a separate message).

Also note the names of the form fields: `tireqty`, `oilqty`, and `sparkqty`. You'll use these names again in the PHP script. Because the names will be reused, it's important to give your form fields meaningful names that you can easily remember when you begin writing the PHP script. Some HTML editors generate field names like `field23` by default. They are difficult to remember. Your life as a PHP programmer will be easier if the names you use reflect the data typed into the field.

You should consider adopting a coding standard for field names so that all field names throughout your site use the same format. This way, you can more easily remember whether, for example, you abbreviated a word in a field name or put in underscores as spaces.

Processing the Form

To process the form, you need to create the script mentioned in the `action` attribute of the `form` tag called `processorder.php`. Open your text editor and create this file. Then type in the following code:

```

<!DOCTYPE html>
<html>
    <head>
        <title>Bob's Auto Parts - Order Results</title>
    </head>
    <body>
        <h1>Bob's Auto Parts</h1>
        <h2>Order Results</h2>
    </body>
</html>

```

Notice how everything you've typed so far is just plain HTML. It's now time to add some simple PHP code to the script.

Embedding PHP in HTML

Under the `<h2>` heading in your file, add the following lines:

```

<?php
    echo '<p>Order processed.</p>';
?>

```

Save the file and load it in your browser by filling out Bob's form and clicking the Submit Order button. You should see something similar to the output shown in Figure 1.2.

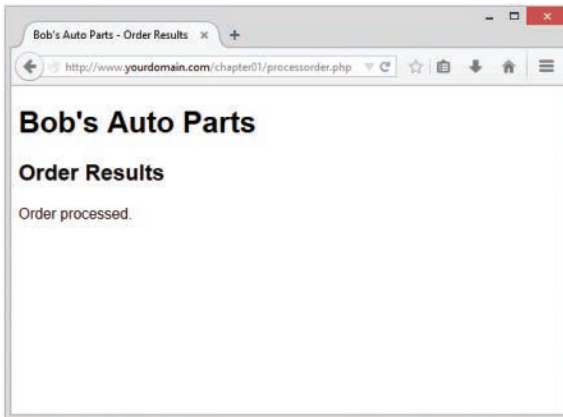


Figure 1.2 Text passed to PHP's `echo` construct is echoed to the browser

Notice how the PHP code you wrote was embedded inside a normal-looking HTML file. Try viewing the source from your browser. You should see this code `<!DOCTYPE html>`

```
<html>
  <head>
    <title>Bob's Auto Parts - Order Results</title>
  </head>
  <body>
    <h1>Bob's Auto Parts</h1>
    <h2>Order Results</h2>
    <p>Order processed.</p>
  </body>
</html>
```

None of the raw PHP is visible because the PHP interpreter has run through the script and replaced it with the output from the script. This means that from PHP you can produce clean HTML viewable with any browser; in other words, the user's browser does not need to understand PHP.

This example illustrates the concept of server-side scripting in a nutshell. The PHP has been interpreted and executed on the web server, as distinct from JavaScript and other client-side technologies interpreted and executed within a web browser on a user's machine.

The code that you now have in this file consists of four types of text:

- HTML
- PHP tags
- PHP statements
- Whitespace

You can also add comments.

Most of the lines in the example are just plain HTML.

PHP Tags

The PHP code in the preceding example began with `<?php` and ended with `?>`. This is similar to all HTML tags because they all begin with a less than (`<`) symbol and end with a greater than (`>`) symbol. These symbols (`<?php` and `?>`) are called *PHP tags*. They tell the web server where the PHP code starts and finishes. Any text between the tags is interpreted as PHP. Any text outside these tags is treated as normal HTML. The PHP tags allow you to *escape* from HTML.

There are actually two styles of PHP tags; each of the following fragments of code is equivalent:

- **XML style**

```
<?php echo '<p>Order processed.</p>'; ?>
```

This is the tag style that we use in this book; it is the preferred PHP tag style. The server administrator cannot turn it off, so you can guarantee it will be available on all servers, which is especially important if you are writing applications that may be used on different installations. This tag style can be used with Extensible Markup Language (XML) documents. In general, we recommend you use this tag style.

- **Short style**

```
<? echo '<p>Order processed.</p>'; ?>
```

This tag style is the simplest and follows the style of a Standard Generalized Markup Language (SGML) processing instruction. To use this type of tag—which is the shortest to type—you either need to enable the `short_open_tag` setting in your config file or compile PHP with short tags enabled. You can find more information on how to use this tag style in Appendix A. The use of this style is not recommended for use in code you plan to distribute. It will not work in many environments as it is no longer enabled by default.

PHP Statements

You tell the PHP interpreter what to do by including PHP statements between your opening and closing tags. The preceding example used only one type of statement:

```
echo '<p>Order processed.</p>';
```

As you have probably guessed, using the `echo` construct has a very simple result: It prints (or echoes) the string passed to it to the browser. In Figure 1.2, you can see the result is that the text `Order processed.` appears in the browser window.

Notice that there is a semicolon at the end of the `echo` statement. Semicolons separate statements in PHP much like periods separate sentences in English. If you have programmed in C or Java before, you will be familiar with using the semicolon in this way.

Leaving off the semicolon is a common syntax error that is easily made. However, it's equally easy to find and to correct.

Whitespace

Spacing characters such as newlines (carriage returns), spaces, and tabs are known as *whitespace*. As you probably already know, browsers ignore whitespace in HTML, and so does the PHP engine. Consider these two HTML fragments:

```
<h1>Welcome to Bob's Auto Parts!</h1><p>What would you like to order today?</p>
```

and

```
<h1>Welcome          to Bob's
Auto Parts!</h1>
<p>What would you like
to order today?</p>
```

These two snippets of HTML code produce identical output because they appear the same to the browser. However, you can and are encouraged to use whitespace sensibly in your HTML as an aid to humans—to enhance the readability of your HTML code. The same is true for PHP. You don't need to have any whitespace between PHP statements, but it makes the code much easier to read if you put each statement on a separate line. For example,

```
echo 'hello ';
echo 'world';
```

and

```
echo 'hello ';echo 'world';
```

are equivalent, but the first version is easier to read.

Comments

Comments are exactly that: Comments in code act as notes to people reading the code. Comments can be used to explain the purpose of the script, who wrote it, why they wrote it the way they did, when it was last modified, and so on. You generally find comments in all but the simplest PHP scripts.

The PHP interpreter ignores any text in comments. Essentially, the PHP parser skips over the comments, making them equivalent to whitespace.

PHP supports C, C++, and shell script-style comments.

The following is a C-style, multiline comment that might appear at the start of a PHP script:

```
/* Author: Bob Smith
   Last modified: April 10
   This script processes the customer orders.
*/
```

Multiline comments should begin with a `/*` and end with `*/`. As in C, multiline comments cannot be nested.

You can also use single-line comments, either in the C++ style:

```
echo '<p>Order processed.</p>'; // Start printing order
```

or in the shell script style:

```
echo '<p>Order processed.</p>'; # Start printing order
```

With both of these styles, everything after the comment symbol (# or //) is a comment until you reach the end of the line or the ending PHP tag, whichever comes first.

In the following line of code, the text before the closing tag, `here is a comment`, is part of a comment. The text after the closing tag, `here is not`, will be treated as HTML because it is outside the closing tag:

```
// here is a comment ?> here is not
```

Adding Dynamic Content

So far, you haven't used PHP to do anything you couldn't have done with plain HTML.

The main reason for using a server-side scripting language is to be able to provide dynamic content to a site's users. This is an important application because content that changes according to users' needs or over time will keep visitors coming back to a site. PHP allows you to do this easily.

Let's start with a simple example. Replace the PHP in `processorder.php` with the following code:

```
<?php
    echo "<p>Order processed at ";
    echo date('H:i, jS F Y');
    echo "</p>";
?>
```

You could also write this on one line, using the concatenation operator (`.`), as

```
<?php
    echo "<p>Order processed at ".date('H:i, jS F Y')."</p>";
?>
```

In this code, PHP's built-in `date()` function tells the customer the date and time when his order was processed. This information will be different each time the script is run. The output of running the script on one occasion is shown in Figure 1.3.

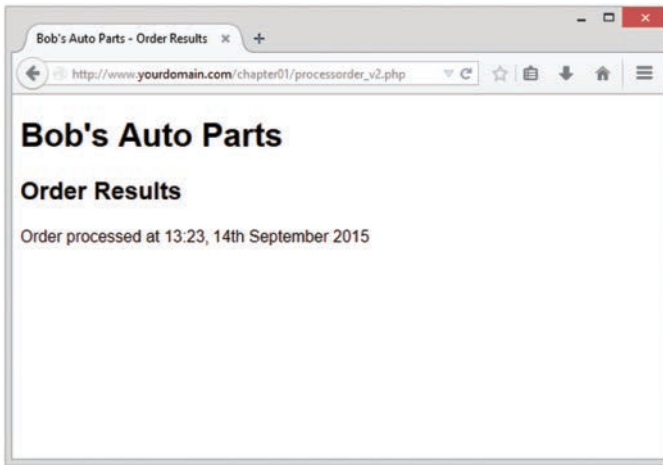


Figure 1.3 PHP's `date()` function returns a formatted date string

Calling Functions

Look at the call to `date()`. This is the general form that function calls take. PHP has an extensive library of functions you can use when developing web applications. Most of these functions need to have some data passed to them and return some data.

Now look at the function call again:

```
date('H:i, jS F')
```

Notice that it passes a string (text data) to the function inside a pair of parentheses. The element within the parentheses is called the function's *argument* or *parameter*. Such arguments are the input the function uses to output some specific results.

Using the `date()` Function

The `date()` function expects the argument you pass it to be a format string, representing the style of output you would like. Each letter in the string represents one part of the date and time. `H` is the hour in a 24-hour format with leading zeros where required, `i` is the minutes with a leading zero where required, `j` is the day of the month without a leading zero, `s` represents the ordinal suffix (in this case `th`), and `F` is the full name of the month.

Note

If `date()` gives you a warning about not having set the timezone, you should add the `date.timezone` setting to your `php.ini` file. More information on this can be found in the sample `php.ini` file in Appendix A.

For a full list of formats supported by `date()`, see Chapter 19, “Managing the Date and Time.”

Accessing Form Variables

The whole point of using the order form is to collect customers’ orders. Getting the details of what the customers typed is easy in PHP, but the exact method depends on the version of PHP you are using and a setting in your `php.ini` file.

Form Variables

Within your PHP script, you can access each form field as a PHP variable whose name relates to the name of the form field. You can recognize variable names in PHP because they all start with a dollar sign (`$`). (Forgetting the dollar sign is a common programming error.)

Depending on your PHP version and setup, you can access the form data via variables in different ways. In recent versions of PHP, all but one of these ways have been deprecated, so beware if you have used PHP in the past that this has changed.

You may access the contents of the field `tireqty` in the following way:

```
$_POST['tireqty']
```

`$_POST` is an array containing data submitted via an HTTP POST request—that is, the form method was set to POST. There are three of these arrays that may contain form data: `$_POST`, `$_GET`, and `$_REQUEST`. One of the `$_GET` or `$_POST` arrays holds the details of all the form variables. Which array is used depends on whether the method used to submit the form was GET or POST, respectively. In addition, a combination of all data submitted via GET or POST is also available through `$_REQUEST`.

If the form was submitted via the POST method, the data entered in the `tireqty` box will be stored in `$_POST['tireqty']`. If the form was submitted via GET, the data will be in `$_GET['tireqty']`. In either case, the data will also be available in `$_REQUEST['tireqty']`.

These arrays are some of the *superglobal* arrays. We will revisit the superglobals when we discuss variable scope later in this chapter.

Let’s look at an example that creates easier-to-use copies of variables.

To copy the value of one variable into another, you use the assignment operator, which in PHP is an equal sign (`=`). The following statement creates a new variable named `$tireqty` and copies the contents of `$_POST['tireqty']` into the new variable:

```
$tireqty = $_POST['tireqty'];
```

Place the following block of code at the start of the processing script. All other scripts in this book that handle data from a form contain a similar block at the start. Because this code

will not produce any output, placing it above or below the `<html>` and other HTML tags that start your page makes no difference. We generally place such blocks at the start of the script to make them easy to find.

```
<?php
    // create short variable names
    $tireqty = $_POST['tireqty'];
    $oilqty = $_POST['oilqty'];
    $sparkqty = $_POST['sparkqty'];
?>
```

This code creates three new variables—`$tireqty`, `$oilqty`, and `$sparkqty`—and sets them to contain the data sent via the `POST` method from the form.

You can output the values of these variables to the browser by doing, for example:

```
echo $tireqty.' tires<br />';
```

However, this approach is not recommended.

At this stage, you have not checked the variable contents to make sure sensible data has been entered in each form field. Try entering deliberately wrong data and observe what happens. After you have read the rest of the chapter, you might want to try adding some data validation to this script.

Taking data directly from the user and outputting it to the browser like this is an extremely risky practice from a security perspective. We do not recommend this approach. You should filter input data. We will start to cover input filtering in Chapter 4, “String Manipulation and Regular Expressions,” and discuss security in depth in Chapter 14, “Web Application Security Risks.”

For now, it’s enough to know that you should echo out user data to the browser after passing it through a function called `htmlspecialchars()`. For example, in this case, we would do the following:

```
echo htmlspecialchars($tireqty).' tires<br />';
```

To make the script start doing something visible, add the following lines to the bottom of your PHP script:

```
echo '<p>Your order is as follows: </p>';
echo htmlspecialchars($tireqty).' tires<br />';
echo htmlspecialchars($oilqty).' bottles of oil<br />';
echo htmlspecialchars($sparkqty).' spark plugs<br />';
```

If you now load this file in your browser, the script output should resemble what is shown in Figure 1.4. The actual values shown, of course, depend on what you typed into the form.

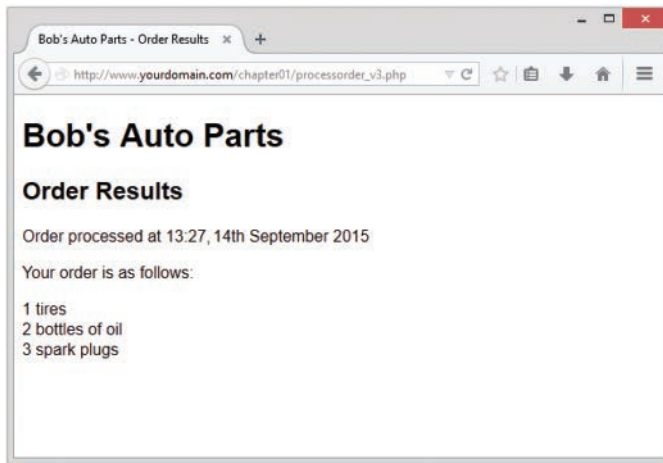


Figure 1.4 The form variables the user typed in are easily accessible in `processor.php`

The following sections describe a couple of interesting elements of this example.

String Concatenation

In the sample script, `echo` prints the value the user typed in each form field, followed by some explanatory text. If you look closely at the `echo` statements, you can see that the variable name and following text have a period (.) between them, such as this:

```
echo htmlspecialchars($tireqty).' tires<br />';
```

This period is the string concatenation operator, which adds strings (pieces of text) together. You will often use it when sending output to the browser with `echo`. This way, you can avoid writing multiple `echo` commands.

You can also place simple variables inside a double-quoted string to be echoed. (Arrays are somewhat more complicated, so we look at combining arrays and strings in Chapter 4.) Consider this example:

```
$tireqty = htmlspecialchars($tireqty);
echo "$tireqty tires<br />";
```

This is equivalent to the first statement shown in this section. Either format is valid, and which one you use is a matter of personal taste. This process, replacing a variable with its contents within a string, is known as *interpolation*.

Note that interpolation is a feature of double-quoted strings only. You cannot place variable names inside a single-quoted string in this way. Running the following line of code

```
echo '$tireqty tires<br />';
```

simply sends `$tireqty tires
` to the browser. Within double quotation marks, the variable name is replaced with its value. Within single quotation marks, the variable name or any other text is sent unaltered.

Variables and Literals

The variables and strings concatenated together in each of the `echo` statements in the sample script are different types of things. Variables are symbols for data. The strings are data themselves. When we use a piece of raw data in a program like this, we call it a *literal* to distinguish it from a *variable*. `$tireQty` is a variable, a symbol that represents the data the customer typed in. On the other hand, `' tires
'` is a literal. You can take it at face value. Well, almost. Remember the second example in the preceding section? PHP replaced the variable name `$tireQty` in the string with the value stored in the variable.

Remember the two kinds of strings mentioned already: ones with double quotation marks and ones with single quotation marks. PHP tries to evaluate strings in double quotation marks, resulting in the behavior shown earlier. Single-quoted strings are treated as true literals.

There is also a third way of specifying strings using the heredoc syntax (`<<<`), which will be familiar to Perl users. Heredoc syntax allows you to specify long strings tidily, by specifying an end marker that will be used to terminate the string. The following example creates a three-line string and echoes it:

```
echo <<<theEnd
    line 1
    line 2
    line 3
theEnd
```

The token `theEnd` is entirely arbitrary. It just needs to be guaranteed not to appear in the text. To close a heredoc string, place a closing token at the start of a line.

Heredoc strings are interpolated, like double-quoted strings.

Understanding Identifiers

Identifiers are the names of variables. (The names of functions and classes are also identifiers; we look at functions and classes in Chapter 5, “Reusing Code and Writing Functions,” and Chapter 6, “Object-Oriented PHP.”) You need to be aware of the simple rules defining valid identifiers:

- Identifiers can be of any length and can consist of letters, numbers, and underscores.
- Identifiers cannot begin with a digit.
- In PHP, identifiers are case sensitive. `$tireQty` is not the same as `$TireQty`. Trying to use them interchangeably is a common programming error. Function names are an exception to this rule: Their names can be used in any case.
- A variable can have the same name as a function. This usage is confusing, however, and should be avoided. Also, you cannot create a function with the same name as another function.

You can declare and use your own variables in addition to the variables you are passed from the HTML form.

One of the features of PHP is that it does not require you to declare variables before using them. A variable is created when you first assign a value to it. See the next section for details.

You assign values to variables using the assignment operator (=) as you did when copying one variable's value to another. On Bob's site, you want to work out the total number of items ordered and the total amount payable. You can create two variables to store these numbers. To begin with, you need to initialize each of these variables to zero by adding these lines to the bottom of your PHP script.

```
$totalqty = 0;
$totalamount = 0.00;
```

Each of these two lines creates a variable and assigns a literal value to it. You can also assign variable values to variables, as shown in this example:

```
$totalqty = 0;
$totalamount = $totalqty;
```

Examining Variable Types

A variable's type refers to the kind of data stored in it. PHP provides a set of data types. Different data can be stored in different data types.

PHP's Data Types

PHP supports the following basic data types:

- **Integer**—Used for whole numbers
- **Float** (also called **double**)—Used for real numbers
- **String**—Used for strings of characters
- **Boolean**—Used for `true` or `false` values
- **Array**—Used to store multiple data items (see Chapter 3, “Using Arrays”)
- **Object**—Used for storing instances of classes (see Chapter 6)

Three special types are also available: `NULL`, `resource`, and `callable`.

Variables that have not been given a value, have been unset, or have been given the specific value `NULL` are of type `NULL`.

Certain built-in functions (such as database functions) return variables that have the type `resource`. They represent external resources (such as database connections). You will almost certainly not directly manipulate a resource variable, but frequently they are returned by functions and must be passed as parameters to other functions.

Callables are essentially functions that are passed to other functions.

Type Strength

PHP is called a weakly typed or dynamically typed language. In most programming languages, variables can hold only one type of data, and that type must be declared before the variable can be used, as in C. In PHP, the type of a variable is determined by the value assigned to it.

For example, when you created `$totalqty` and `$totalamount`, their initial types were determined as follows:

```
$totalqty = 0;
$totalamount = 0.00;
```

Because you assigned 0, an integer, to `$totalqty`, this is now an integer type variable. Similarly, `$totalamount` is now of type float.

Strangely enough, you could now add a line to your script as follows:

```
$totalamount = 'Hello';
```

The variable `$totalamount` would then be of type string. PHP changes the variable type according to what is stored in it at any given time.

This ability to change types transparently on the fly can be extremely useful. Remember PHP “automagically” knows what data type you put into your variable. It returns the data with the same data type when you retrieve it from the variable.

Type Casting

You can pretend that a variable or value is of a different type by using a type cast. This feature works identically to the way it works in C. You simply put the temporary type in parentheses in front of the variable you want to cast.

For example, you could have declared the two variables from the preceding section using a cast:

```
$totalqty = 0;
$totalamount = (float)$totalqty;
```

The second line means “Take the value stored in `$totalqty`, interpret it as a float, and store it in `$totalamount`.” The `$totalamount` variable will be of type float. The cast variable does not change types, so `$totalqty` remains of type integer.

You can also use built-in functions to test and set type, which you will learn about later in this chapter.

Variable Variables

PHP provides one other type of variable: the variable variable. Variable variables enable you to change the name of a variable dynamically.

As you can see, PHP allows a lot of freedom in this area. All languages enable you to change the value of a variable, but not many allow you to change the variable’s type, and even fewer allow you to change the variable’s name.

A variable variable works by using the value of one variable as the name of another. For example, you could set

```
$varname = 'tireqty';
```

You can then use `$$varname` in place of `$tireqty`. For example, you can set the value of `$tireqty` as follows:

```
$$varname = 5;
```

This is equivalent to

```
$tireqty = 5;
```

This approach might seem somewhat obscure, but we'll revisit its use later. Instead of having to list and use each form variable separately, you can use a loop and variable variable to process them all automatically. You can find an example illustrating this in the section on `for` loops later in this chapter.

Declaring and Using Constants

As you saw previously, you can readily change the value stored in a variable. You can also declare constants. A constant stores a value just like a variable, but its value is set once and then cannot be changed elsewhere in the script.

In the sample application, you might store the prices for each item on sale as a constant. You can define these constants using the `define` function:

```
define('TIREPRICE', 100);
define('OILPRICE', 10);
define('SPARKPRICE', 4);
```

Now add these lines of code to your script. You now have three constants that can be used to calculate the total of the customer's order.

Notice that the names of the constants appear in uppercase. This convention, borrowed from C, makes it easy to distinguish between variables and constants at a glance. Following this convention is not required but will make your code easier to read and maintain.

One important difference between constants and variables is that when you refer to a constant, it does not have a dollar sign in front of it. If you want to use the value of a constant, use its name only. For example, to use one of the constants just created, you could type

```
echo TIREPRICE;
```

As well as the constants you define, PHP sets a large number of its own. An easy way to obtain an overview of them is to run the `phpinfo()` function:

```
phpinfo();
```

This function provides a list of PHP's predefined variables and constants, among other useful information. We will discuss some of them as we go along.

One other difference between variables and constants is that constants can store only boolean, integer, float, or string data. These types are collectively known as scalar values.

Understanding Variable Scope

The term *scope* refers to the places within a script where a particular variable is visible.

The six basic scope rules in PHP are as follows:

- Built-in superglobal variables are visible everywhere within a script.
- Constants, once declared, are always visible globally; that is, they can be used inside and outside functions.
- Global variables declared in a script are visible throughout that script, but *not inside functions*.
- Variables inside functions that are declared as global refer to the global variables of the same name.
- Variables created inside functions and declared as static are invisible from outside the function but keep their value between one execution of the function and the next. (We explain this idea fully in Chapter 5.)
- Variables created inside functions are local to the function and cease to exist when the function terminates.

The arrays `$_GET` and `$_POST` and some other special variables have their own scope rules.

They are known as *superglobals* and can be seen everywhere, both inside and outside functions.

The complete list of superglobals is as follows:

- `$GLOBALS`—An array of all global variables (Like the `global` keyword, this allows you to access global variables inside a function—for example, as `$GLOBALS['myvariable']`.)
- `$_SERVER`—An array of server environment variables
- `$_GET`—An array of variables passed to the script via the `GET` method
- `$_POST`—An array of variables passed to the script via the `POST` method
- `$_COOKIE`—An array of cookie variables
- `$_FILES`—An array of variables related to file uploads
- `$_ENV`—An array of environment variables
- `$_REQUEST`—An array of all user input including the contents of input including `$_GET`, `$_POST`, and `$_COOKIE` (but not including `$_FILES`)
- `$_SESSION`—An array of session variables

We come back to each of these superglobals throughout the book as they become relevant.

We cover scope in more detail when we discuss functions and classes later in this chapter. For the time being, all the variables we use are global by default.

Using Operators

Operators are symbols that you can use to manipulate values and variables by performing an operation on them. You need to use some of these operators to work out the totals and tax on the customer's order.

We've already mentioned two operators: the assignment operator (=) and the string concatenation operator (.). In the following sections, we describe the complete list.

In general, operators can take one, two, or three arguments, with the majority taking two. For example, the assignment operator takes two: the storage location on the left side of the = symbol and an expression on the right side. These arguments are called *operands*—that is, the things that are being operated upon.

Arithmetic Operators

Arithmetic operators are straightforward; they are just the normal mathematical operators. PHP's arithmetic operators are shown in Table 1.1.

Table 1.1 PHP's Arithmetic Operators

Operator	Name	Example
+	Addition	<code>\$a + \$b</code>
-	Subtraction	<code>\$a - \$b</code>
*	Multiplication	<code>\$a * \$b</code>
/	Division	<code>\$a / \$b</code>
%	Modulus	<code>\$a % \$b</code>

With each of these operators, you can store the result of the operation, as in this example:

```
$result = $a + $b;
```

Addition and subtraction work as you would expect. The result of these operators is to add or subtract, respectively, the values stored in the `$a` and `$b` variables.

You can also use the subtraction symbol (-) as a unary operator—that is, an operator that takes one argument or operand—to indicate negative numbers, as in this example:

```
$a = -1;
```

Multiplication and division also work much as you would expect. Note the use of the asterisk as the multiplication operator rather than the regular multiplication symbol, and the forward slash as the division operator rather than the regular division symbol.

The modulus operator returns the remainder calculated by dividing the `$a` variable by the `$b` variable. Consider this code fragment:

```
$a = 27;  
$b = 10;  
$result = $a % $b;
```

The value stored in the `$result` variable is the remainder when you divide 27 by 10—that is, 7.

You should note that arithmetic operators are usually applied to integers or doubles. If you apply them to strings, PHP will try to convert the string to a number. If it contains an `e` or an `E`, it will be read as being in scientific notation and converted to a float; otherwise, it will be converted to an integer. PHP will look for digits at the start of the string and use them as the value; if there are none, the value of the string will be zero.

String Operators

You’ve already seen and used the only string operator. You can use the string concatenation operator to add two strings and to generate and store a result much as you would use the addition operator to add two numbers:

```
$a = "Bob's ";
$b = "Auto Parts";
$result = $a.$b;
```

The `$result` variable now contains the string "Bob's Auto Parts".

Assignment Operators

You’ve already seen the basic assignment operator (`=`). Always refer to this as the assignment operator and read it as “is set to.” For example,

```
$totalqty = 0;
```

This line should be read as “`$totalqty` is set to zero.” We explain why when we discuss the comparison operators later in this chapter, but if you call it equals, you will get confused.

Values Returned from Assignment

Using the assignment operator returns an overall value similar to other operators. If you write

```
$a + $b
```

the value of this expression is the result of adding the `$a` and `$b` variables together. Similarly, you can write

```
$a = 0;
```

The value of this whole expression is zero.

This technique enables you to form expressions such as

```
$b = 6 + ($a = 5);
```

This line sets the value of the `$b` variable to 11. This behavior is generally true of assignments: The value of the whole assignment statement is the value that is assigned to the left operand.

When working out the value of an expression, you can use parentheses to increase the precedence of a subexpression, as shown here. This technique works exactly the same way as in mathematics.

Combined Assignment Operators

In addition to the simple assignment, there is a set of combined assignment operators. Each of them is a shorthand way of performing another operation on a variable and assigning the result back to that variable. For example,

```
$a += 5;
```

This is equivalent to writing

```
$a = $a + 5;
```

Combined assignment operators exist for each of the arithmetic operators and for the string concatenation operator. A summary of all the combined assignment operators and their effects is shown in Table 1.2.

Table 1.2 PHP's Combined Assignment Operators

Operator	Use	Equivalent To
+=	<code>\$a += \$b</code>	<code>\$a = \$a + \$b</code>
-=	<code>\$a -= \$b</code>	<code>\$a = \$a - \$b</code>
*=	<code>\$a *= \$b</code>	<code>\$a = \$a * \$b</code>
/=	<code>\$a /= \$b</code>	<code>\$a = \$a / \$b</code>
%=	<code>\$a %= \$b</code>	<code>\$a = \$a % \$b</code>
.=	<code>\$a .= \$b</code>	<code>\$a = \$a . \$b</code>

Pre- and Post-Increment and Decrement

The pre- and post-increment (++) and decrement (--) operators are similar to the += and -= operators, but with a couple of twists.

All the increment operators have two effects: They increment and assign a value. Consider the following:

```
$a=4;
echo ++$a;
```

The second line uses the pre-increment operator, so called because the ++ appears before the \$a. This has the effect of first incrementing \$a by 1 and second, returning the incremented value. In this case, \$a is incremented to 5, and then the value 5 is returned and printed. The value of this whole expression is 5. (Notice that the actual value stored in \$a is changed: It is not just returning \$a + 1.)

If the ++ is after the \$a, however, you are using the post-increment operator. It has a different effect. Consider the following:

```
$a=4;
echo $a++;
```

In this case, the effects are reversed. That is, first, the value of `$a` is returned and printed, and second, it is incremented. The value of this whole expression is 4. This is the value that will be printed. However, the value of `$a` after this statement is executed is 5.

As you can probably guess, the behavior is similar for the `--` (decrement) operator. However, the value of `$a` is decremented instead of being incremented.

Reference Operator

The reference operator (`&`, an ampersand) can be used in conjunction with assignment. Normally, when one variable is assigned to another, a copy is made of the first variable and stored elsewhere in memory. For example,

```
$a = 5;
$b = $a;
```

These code lines make a second copy of the value in `$a` and store it in `$b`. If you subsequently change the value of `$a`, `$b` will not change:

```
$a = 7; // $b will still be 5
```

You can avoid making a copy by using the reference operator. For example,

```
$a = 5;
$b = &$a;
$a = 7; // $a and $b are now both 7
```

References can be a bit tricky. Remember that a reference is like an alias rather than like a pointer. Both `$a` and `$b` point to the same piece of memory. You can change this by unsetting one of them as follows:

```
unset($a);
```

Unsetting does not change the value of `$b` (7) but does break the link between `$a` and the value 7 stored in memory.

Comparison Operators

The comparison operators compare two values. Expressions using these operators return either of the logical values `true` or `false` depending on the result of the comparison.

The Equal Operator

The equal comparison operator (`==`, two equal signs) enables you to test whether two values are equal. For example, you might use the expression

```
$a == $b
```

to test whether the values stored in `$a` and `$b` are the same. The result returned by this expression is `true` if they are equal or `false` if they are not.

You might easily confuse `==` with `=`, the assignment operator. Using the wrong operator will work without giving an error but generally will not give you the result you wanted. In general,

nonzero values evaluate to `true` and zero values to `false`. Say that you have initialized two variables as follows:

```
$a = 5;
$b = 7;
```

If you then test `$a = $b`, the result will be `true`. Why? The value of `$a = $b` is the value assigned to the left side, which in this case is 7. Because 7 is a nonzero value, the expression evaluates to `true`. If you intended to test `$a == $b`, which evaluates to `false`, you have introduced a logic error in your code that can be extremely difficult to find. Always check your use of these two operators and check that you have used the one you intended to use.

Using the assignment operator rather than the equals comparison operator is an easy mistake to make, and you will probably make it many times in your programming career.

Other Comparison Operators

PHP also supports a number of other comparison operators. A summary of all the comparison operators is shown in Table 1.3. One to note is the identical operator (`===`), which returns `true` only if the two operands are both equal and of the same type. For example, `0=='0'` will be `true`, but `0=== '0'` will not because one zero is an integer and the other zero is a string.

Table 1.3 PHP's Comparison Operators

Operator	Name	Use
<code>==</code>	Equals	<code>\$a == \$b</code>
<code>===</code>	Identical	<code>\$a === \$b</code>
<code>!=</code>	Not equal	<code>\$a != \$b</code>
<code>!==</code>	Not identical	<code>\$a !== \$b</code>
<code><></code>	Not equal (comparison operator)	<code>\$a <> \$b</code>
<code><</code>	Less than	<code>\$a < \$b</code>
<code>></code>	Greater than (comparison operator)	<code>\$a > \$b</code>
<code><=</code>	Less than or equal to	<code>\$a <= \$b</code>
<code>>=</code>	Greater than or equal to	<code>\$a >= \$b</code>

Logical Operators

The logical operators combine the results of logical conditions. For example, you might be interested in a case in which the value of a variable, `$a`, is between 0 and 100. You would need to test both the conditions `$a >= 0` and `$a <= 100`, using the AND operator, as follows:

```
$a >= 0 && $a <= 100
```

PHP supports logical AND, OR, XOR (exclusive or), and NOT.

The set of logical operators and their use is summarized in Table 1.4.

Table 1.4 PHP's Logical Operators

Operator	Name	Use	Result
!	NOT	!\$b	Returns true if \$b is false and vice versa
&&	AND	\$a && \$b	Returns true if both \$a and \$b are true; otherwise false
	OR	\$a \$b	Returns true if either \$a or \$b or both are true; otherwise false
and	AND	\$a and \$b	Same as &&, but with lower precedence
or	OR	\$a or \$b	Same as , but with lower precedence
xor	XOR	\$a x or \$b	Returns true if either \$a or \$b is true, and false if they are both true or both false.

The `and` and `or` operators have lower precedence than the `&&` and `||` operators. We cover precedence in more detail later in this chapter.

Bitwise Operators

The bitwise operators enable you to treat an integer as the series of bits used to represent it. You probably will not find a lot of use for the bitwise operators in PHP, but a summary is shown in Table 1.5.

Table 1.5 PHP's Bitwise Operators

Operator	Name	Use	Result
&	Bitwise AND	\$a & \$b	Bits set in \$a and \$b are set in the result.
	Bitwise OR	\$a \$b	Bits set in \$a or \$b are set in the result.
~	Bitwise NOT	~\$a	Bits set in \$a are not set in the result and vice versa.
^	Bitwise XOR	\$a ^ \$b	Bits set in \$a or \$b but not in both are set in the result.
<<	Left shift	\$a << \$b	Shifts \$a left \$b bits.
>>	Right shift	\$a >> \$b	Shifts \$a right \$b bits.

Other Operators

In addition to the operators we have covered so far, you can use several others.

The comma operator (,) separates function arguments and other lists of items. It is normally used incidentally.

Two special operators, `new` and `->`, are used to instantiate a class and access class members, respectively. They are covered in detail in Chapter 6.

There are a few others that we discuss briefly here.

The Ternary Operator

The ternary operator (`?:`) takes the following form:

```
condition ? value if true : value if false
```

This operator is similar to the expression version of an `if-else` statement, which is covered later in this chapter.

A simple example is

```
($grade >= 50 ? 'Passed' : 'Failed')
```

This expression evaluates student grades to `'Passed'` or `'Failed'`.

The Error Suppression Operator

The error suppression operator (`@`) can be used in front of any expression—that is, anything that generates or has a value. For example,

```
$a = @(57/0);
```

Without the `@` operator, this line generates a divide-by-zero warning. With the operator included, the error is suppressed.

If you are suppressing warnings in this way, you need to write some error handling code to check when a warning has occurred. If you have PHP set up with the `track_errors` feature enabled in `php.ini`, the error message will be stored in the global variable `$php_errormsg`.

The Execution Operator

The execution operator is really a pair of operators—a pair of backticks (```) in fact. The backtick is not a single quotation mark; it is usually located on the same key as the `~` (tilde) symbol on your keyboard.

PHP attempts to execute whatever is contained between the backticks as a command at the server's command line. The value of the expression is the output of the command.

For example, under Unix-like operating systems, you can use

```
$out = `ls -la`;
echo '<pre>'.$out.'</pre>';
```

Or, equivalently on a Windows server, you can use

```
$out = `dir c:`;
echo '<pre>'.$out.'</pre>';
```

Either version obtains a directory listing and stores it in `$out`. It can then be echoed to the browser or dealt with in any other way.

There are other ways of executing commands on the server. We cover them in Chapter 17, “Interacting with the File System and the Server.”

Array Operators

There are a number of array operators. The array element operators (`[]`) enable you to access array elements. You can also use the `=>` operator in some array contexts. These operators are covered in Chapter 3.

You also have access to a number of other array operators. We cover them in detail in Chapter 3 as well, but we included them here in Table 1.6 for completeness.

Table 1.6 PHP's Array Operators

Operator	Name	Use	Result
<code>+</code>	Union	<code>\$a + \$b</code>	Returns an array containing everything in <code>\$a</code> and <code>\$b</code>
<code>==</code>	Equality	<code>\$a == \$b</code>	Returns <code>true</code> if <code>\$a</code> and <code>\$b</code> have the same key and value pairs
<code>===</code>	Identity	<code>\$a === \$b</code>	Returns <code>true</code> if <code>\$a</code> and <code>\$b</code> have the same key and value pairs in the same order and of the same type.
<code>!=</code>	Inequality	<code>\$a != \$b</code>	Returns <code>true</code> if <code>\$a</code> and <code>\$b</code> are not equal
<code><></code>	Inequality	<code>\$a <> \$b</code>	Returns <code>true</code> if <code>\$a</code> and <code>\$b</code> are not equal
<code>!==</code>	Non-identity	<code>\$a !== \$b</code>	Returns <code>true</code> if <code>\$a</code> and <code>\$b</code> are not identical

You will notice that the array operators in Table 1.6 all have equivalent operators that work on scalar variables. As long as you remember that `+` performs addition on scalar types and union on arrays—even if you have no interest in the set arithmetic behind that behavior—the behaviors should make sense. You cannot usefully compare arrays to scalar types.

The Type Operator

There is one type operator: `instanceof`. This operator is used in object-oriented programming, but we mention it here for completeness. (Object-oriented programming is covered in Chapter 6.)

The `instanceof` operator allows you to check whether an object is an instance of a particular class, as in this example:

```
class sampleClass{};
$myObject = new sampleClass();
if ($myObject instanceof sampleClass)
    echo "myObject is an instance of sampleClass";
```


Working Out the Form Totals

Now that you know how to use PHP's operators, you are ready to work out the totals and tax on Bob's order form. To do this, add the following code to the bottom of your PHP script:

```
$totalqty = 0;
$totalqty = $tireqty + $oilqty + $sparkqty;
echo "<p>Items ordered: ".$totalqty."<br />";
$totalamount = 0.00;

define('TIREPRICE', 100);
define('OILPRICE', 10);
define('SPARKPRICE', 4);

$totalamount = $tireqty * TIREPRICE
               + $oilqty * OILPRICE
               + $sparkqty * SPARKPRICE;

echo "Subtotal: $".number_format($totalamount,2)."<br />";

$taxrate = 0.10; // local sales tax is 10%
$totalamount = $totalamount * (1 + $taxrate);
echo "Total including tax: $".number_format($totalamount,2)."</p>";
```

If you refresh the page in your browser window, you should see output similar to Figure 1.5.

As you can see, this piece of code uses several operators. It uses the addition (+) and multiplication (*) operators to work out the amounts and the string concatenation operator (.) to set up the output to the browser.

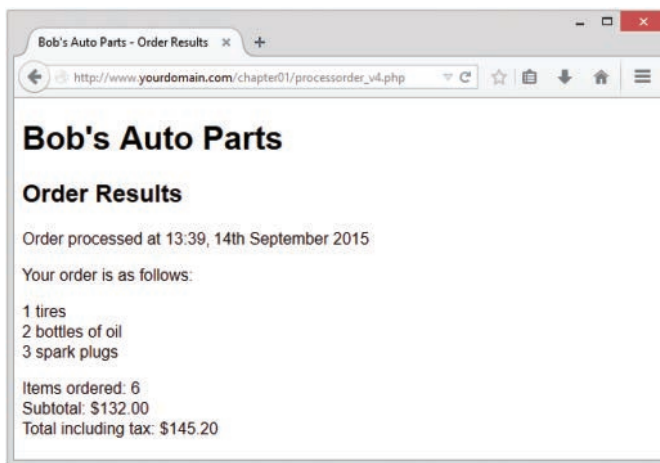


Figure 1.5 The totals of the customer's order have been calculated, formatted, and displayed

It also uses the `number_format()` function to format the totals as strings with two decimal places. This is a function from PHP's Math library.

If you look closely at the calculations, you might ask why the calculations were performed in the order they were. For example, consider this statement:

```
$totalamount = $tireqty * TIREPRICE
               + $oilqty * OILPRICE
               + $sparkqty * SPARKPRICE;
```

The total amount seems to be correct, but why were the multiplications performed before the additions? The answer lies in the precedence of the operators—that is, the order in which they are evaluated.

Understanding Precedence and Associativity

In general, operators have a set precedence, or order, in which they are evaluated. Operators also have associativity, which is the order in which operators of the same precedence are evaluated. This order is generally left to right (called *left* for short), right to left (called *right* for short), or *not relevant*.

Table 1.7 shows operator precedence and associativity in PHP. In this table, operators with the lowest precedence are at the top, and precedence increases as you go down the table.

Table 1.7 Operator Precedence in PHP

Associativity	Operators
left	,
left	Or
left	Xor
left	And
right	Print
left	= += -= *= /= .= %= &= = ^= ~= <<= >>=
left	? :
left	
left	&&
left	
left	^
left	&
n/a	== != === !==

Associativity	Operators
n/a	< <= > >=
left	<< >>
left	+ - .
left	* / %
right	!
n/a	Instanceof
right	~ (int) (float) (string) (array) (object) (bool) @
n/a	++ --
right	[]
n/a	clone new
n/a	()

Notice that we haven't yet covered the operator with the highest precedence: plain old parentheses. The effect of using parentheses is to raise the precedence of whatever is contained within them. This is how you can deliberately manipulate or work around the precedence rules when you need to.

Remember this part of the preceding example:

```
$totalamount = $totalamount * (1 + $taxrate);
```

If you had written

```
$totalamount = $totalamount * 1 + $taxrate;
```

the multiplication operation, having higher precedence than the addition operation, would be performed first, giving an incorrect result. By using the parentheses, you can force the subexpression `1 + $taxrate` to be evaluated first.

You can use as many sets of parentheses as you like in an expression. The innermost set of parentheses is evaluated first.

Also note one other operator in this table we have not yet covered: the `print` language construct, which is equivalent to `echo`. Both constructs generate output.

We generally use `echo` in this book, but you can use `print` if you find it more readable. Neither `print` nor `echo` is really a function, but both can be called as a function with parameters in parentheses. Both can also be treated as an operator: You simply place the string to work with after the keyword `echo` or `print`.

Calling `print` as a function causes it to return a value (1). This capability might be useful if you want to generate output inside a more complex expression but does mean that `print` is marginally slower than `echo`.

Using Variable Handling Functions

Before we leave the world of variables and operators, let's look at PHP's variable handling functions. PHP provides a library of functions that enable you to manipulate and test variables in different ways.

Testing and Setting Variable Types

Most of the variable functions are related to testing the type of function. The two most general are `gettype()` and `settype()`. They have the following function prototypes; that is, this is what arguments expect and what they return:

```
string gettype(mixed var);  
bool settype(mixed var, string type);
```

To use `gettype()`, you pass it a variable. It determines the type and returns a string containing the type name: `bool`, `int`, `double` (for floats, confusingly, for historical reasons), `string`, `array`, `object`, `resource`, or `NULL`. It returns `unknown` type if it is not one of the standard types.

To use `settype()`, you pass it a variable for which you want to change the type and a string containing the new type for that variable from the previous list.

Note

This book and the `php.net` documentation refer to the data type “mixed.” There is no such data type, but because PHP is so flexible with type handling, many functions can take many (or any) data types as an argument. Arguments for which many types are permitted are shown with the pseudo-type “mixed.”

You can use these functions as follows:

```
$a = 56;  
echo gettype($a). '<br />';  
settype($a, 'float');  
echo gettype($a). '<br />';
```

When `gettype()` is called the first time, the type of `$a` is integer. After the call to `settype()`, the type is changed to `float`, which is reported as `double`. (Be aware of this difference.)

PHP also provides some specific type-testing functions. Each takes a variable as an argument and returns either `true` or `false`. The functions are

- `is_array()`—Checks whether the variable is an array
- `is_double()`, `is_float()`, `is_real()` (All the same function)—Checks whether the variable is a float
- `is_long()`, `is_int()`, `is_integer()` (All the same function)—Checks whether the variable is an integer

- `is_string()`—Checks whether the variable is a string
- `is_bool()`—Checks whether the variable is a boolean
- `is_object()`—Checks whether the variable is an object
- `is_resource()`—Checks whether the variable is a resource
- `is_null()`—Checks whether the variable is null
- `is_scalar()`—Checks whether the variable is a scalar—that is, an integer, boolean, string, or float
- `is_numeric()`—Checks whether the variable is any kind of number or a numeric string
- `is_callable()`—Checks whether the variable is the name of a valid function

Testing Variable Status

PHP has several functions for testing the status of a variable. The first is `isset()`, which has the following prototype:

```
bool isset(mixed var[, mixed var[,...]])
```

This function takes a variable name as an argument and returns `true` if it exists and `false` otherwise. You can also pass in a comma-separated list of variables, and `isset()` will return `true` if all the variables are set.

You can wipe a variable out of existence by using its companion function, `unset()`, which has the following prototype:

```
void unset(mixed var[, mixed var[,...]])
```

This function gets rid of the variable it is passed.

The `empty()` function checks to see whether a variable exists and has a nonempty, nonzero value; it returns `true` or `false` accordingly. It has the following prototype:

```
bool empty(mixed var)
```

Let's look at an example using these three functions.

Try adding the following code to your script temporarily:

```
echo 'isset($tireqty): ' .isset($tireqty).'<br />';
echo 'isset($nothere): ' .isset($nothere).'<br />';
echo 'empty($tireqty): ' .empty($tireqty).'<br />';
echo 'empty($nothere): ' .empty($nothere).'<br />';
```

Refresh the page to see the results.

The variable `$tireqty` should return 1 (`true`) from `isset()` regardless of what value you entered in that form field and regardless of whether you entered a value at all. Whether it is `empty()` depends on what you entered in it.

The variable `$nothere` does not exist, so it generates a blank (`false`) result from `isset()` and a `1 (true)` result from `empty()`.

These functions are handy when you need to make sure that the user filled out the appropriate fields in the form.

Reinterpreting Variables

You can achieve the equivalent of casting a variable by calling a function. The following three functions can be useful for this task:

```
int intval(mixed var[, int base=10])
float floatval(mixed var)
string strval(mixed var)
```

Each accepts a variable as input and returns the variable's value converted to the appropriate type. The `intval()` function also allows you to specify the base for conversion when the variable to be converted is a string. (This way, you can convert, for example, hexadecimal strings to integers.)

Making Decisions with Conditionals

Control structures are the structures within a language that allow you to control the flow of execution through a program or script. You can group them into conditional (or branching) structures and repetition structures (or loops).

If you want to sensibly respond to your users' input, your code needs to be able to make decisions. The constructs that tell your program to make decisions are called *conditionals*.

if Statements

You can use an `if` statement to make a decision. You should give the `if` statement a condition to use. If the condition is `true`, the following block of code will be executed. Conditions in `if` statements must be surrounded by parentheses `()`.

For example, if a visitor orders no tires, no bottles of oil, and no spark plugs from Bob, it is probably because she accidentally clicked the Submit Order button before she had finished filling out the form. Rather than telling the visitor "Order processed," the page could give her a more useful message.

When the visitor orders no items, you might like to say, "You did not order anything on the previous page!" You can do this easily by using the following `if` statement:

```
if ($totalqty == 0)
    echo 'You did not order anything on the previous page!<br />';
```

The condition you are using here is `$totalqty == 0`. Remember that the equals operator (`==`) behaves differently from the assignment operator (`=`).

The condition `$totalqty == 0` will be `true` if `$totalqty` is equal to zero. If `$totalqty` is not equal to zero, the condition will be `false`. When the condition is `true`, the `echo` statement will be executed.

Code Blocks

Often you may have more than one statement you want executed according to the actions of a conditional statement such as `if`. You can group a number of statements together as a *block*. To declare a block, you enclose it in curly braces:

```
if ($totalqty == 0) {
    echo '<p style="color:red">';
    echo 'You did not order anything on the previous page!';
    echo '</p>';
}
```

The three lines enclosed in curly braces are now a block of code. When the condition is `true`, all three lines are executed. When the condition is `false`, all three lines are ignored.

Note

As already mentioned, PHP does not care how you lay out your code. However, you should indent your code for readability purposes. Indenting is used to enable you to see at a glance which lines will be executed only if conditions are met, which statements are grouped into blocks, and which statements are parts of loops or functions. In the previous examples, you can see that the statement depending on the `if` statement and the statements making up the block are indented.

else Statements

You may often need to decide not only whether you want an action performed, but also which of a set of possible actions you want performed.

An `else` statement allows you to define an alternative action to be taken when the condition in an `if` statement is `false`. Say you want to warn Bob's customers when they do not order anything. On the other hand, if they do make an order, instead of a warning, you want to show them what they ordered.

If you rearrange the code and add an `else` statement, you can display either a warning or a summary:

```
if ($totalqty == 0) {
    echo "You did not order anything on the previous page!<br />";
} else {
    echo htmlspecialchars($tireqty).' tires<br />';
    echo htmlspecialchars($oilqty).' bottles of oil<br />';
    echo htmlspecialchars($sparkqty).' spark plugs<br />';
}
```

You can build more complicated logical processes by nesting `if` statements within each other. In the following code, the summary will be displayed only if the condition `$totalqty == 0` is true, and each line in the summary will be displayed only if its own condition is met:

```
if ($totalqty == 0) {
    echo "You did not order anything on the previous page!<br />";
} else {
    if ($tireqty > 0)
        echo htmlspecialchars($tireqty).' tires<br />';
    if ($oilqty > 0)
        echo htmlspecialchars($oilqty).' bottles of oil<br />';
    if ($sparkqty > 0)
        echo htmlspecialchars($sparkqty).' spark plugs<br />';
}
```

elseif Statements

For many of the decisions you make, you have more than two options. You can create a sequence of many options using the `elseif` statement, which is a combination of an `else` and an `if` statement. When you provide a sequence of conditions, the program can check each until it finds one that is true.

Bob provides a discount for large orders of tires. The discount scheme works like this:

- Fewer than 10 tires purchased—No discount
- 10–49 tires purchased—5% discount
- 50–99 tires purchased—10% discount
- 100 or more tires purchased—15% discount

You can create code to calculate the discount using conditions and `if` and `elseif` statements. In this case, you need to use the AND operator (`&&`) to combine two conditions into one:

```
if ($tireqty < 10) {
    $discount = 0;
} elseif (($tireqty >= 10) && ($tireqty <= 49)) {
    $discount = 5;
} elseif (($tireqty >= 50) && ($tireqty <= 99)) {
    $discount = 10;
} elseif ($tireqty >= 100) {
    $discount = 15;
}
```

Note that you are free to type `elseif` or `else if`—versions with or without a space are both correct.

If you are going to write a cascading set of `elseif` statements, you should be aware that only one of the blocks or statements will be executed. It did not matter in this example because

all the conditions were mutually exclusive; only one can be true at a time. If you write conditions in a way that more than one could be true at the same time, only the block or statement following the first true condition will be executed.

switch Statements

The `switch` statement works in a similar way to the `if` statement, but it allows the condition to take more than two values. In an `if` statement, the condition can be either `true` or `false`. In a `switch` statement, the condition can take any number of different values, as long as it evaluates to a simple type (integer, string, or float). You need to provide a `case` statement to handle each value you want to react to and, optionally, a default case to handle any that you do not provide a specific `case` statement for.

Bob wants to know what forms of advertising are working for him, so you can add a question to the order form. Insert this HTML into the order form, and the form will resemble Figure 1.6:

```
<tr>
  <td>How did you find Bob's?</td>
  <td><select name="find">
    <option value = "a">I'm a regular customer</option>
    <option value = "b">TV advertising</option>
    <option value = "c">Phone directory</option>
    <option value = "d">Word of mouth</option>
  </select>
</td>
</tr>
```

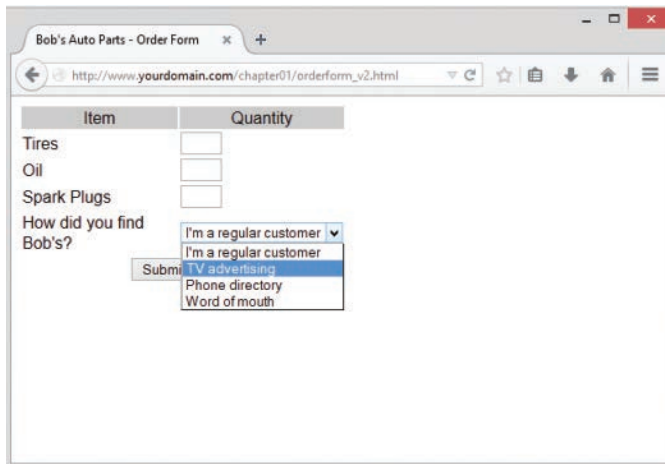


Figure 1.6 The order form now asks visitors how they found Bob's Auto Parts

This HTML code adds a new form variable (called `find`) whose value will be 'a', 'b', 'c', or 'd'. You could handle this new variable with a series of `if` and `elseif` statements like this:

```
if ($find == "a") {
    echo "<p>Regular customer.</p>";
} elseif ($find == "b") {
    echo "<p>Customer referred by TV advert.</p>";
} elseif ($find == "c") {
    echo "<p>Customer referred by phone directory.</p>";
} elseif ($find == "d") {
    echo "<p>Customer referred by word of mouth.</p>";
} else {
    echo "<p>We do not know how this customer found us.</p>";
}
```

Alternatively, you could write a `switch` statement:

```
switch($find) {
    case "a" :
        echo "<p>Regular customer.</p>";
        break;
    case "b" :
        echo "<p>Customer referred by TV advert.</p>";
        break;
    case "c" :
        echo "<p>Customer referred by phone directory.</p>";
        break;
    case "d" :
        echo "<p>Customer referred by word of mouth.</p>";
        break;
    default :
        echo "<p>We do not know how this customer found us.</p>";
        break;
}
```

(Note that both of these examples assume you have extracted `$find` from the `$_POST` array.)

The `switch` statement behaves somewhat differently from an `if` or `elseif` statement. An `if` statement affects only one statement unless you deliberately use curly braces to create a block of statements. A `switch` statement behaves in the opposite way. When a `case` statement in a `switch` is activated, PHP executes statements until it reaches a `break` statement. Without `break` statements, a `switch` would execute all the code following the `case` that was true. When a `break` statement is reached, the next line of code after the `switch` statement is executed.

Comparing the Different Conditionals

If you are not familiar with the statements described in the preceding sections, you might be asking, "Which one is the best?"

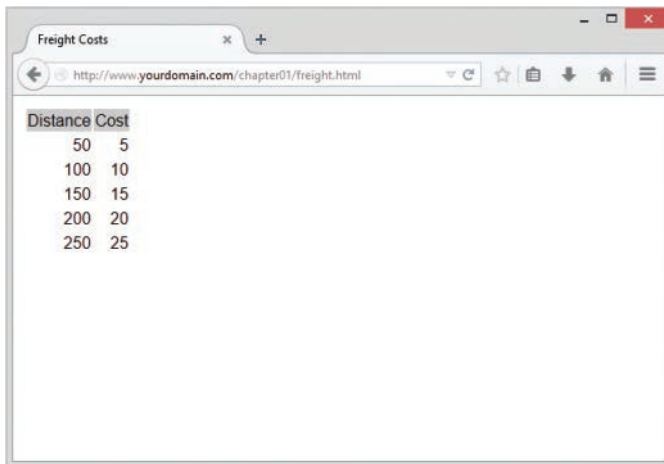
That is not really a question we can answer. There is nothing that you can do with one or more `else`, `elseif`, or `switch` statements that you cannot do with a set of `if` statements. You should try to use whichever conditional will be most readable in your situation. You will acquire a feel for which suits different situations as you gain experience.

Repeating Actions Through Iteration

One thing that computers have always been very good at is automating repetitive tasks. If you need something done the same way a number of times, you can use a loop to repeat some parts of your program.

Bob wants a table displaying the freight cost that will be added to a customer's order. With the courier Bob uses, the cost of freight depends on the distance the parcel is being shipped. This cost can be worked out with a simple formula.

You want the freight table to resemble the table in Figure 1.7.



Distance	Cost
50	5
100	10
150	15
200	20
250	25

Figure 1.7 This table shows the cost of freight as distance increases

Listing 1.2 shows the HTML that displays this table. You can see that it is long and repetitive.

Listing 1.2 **freight.html**—HTML for Bob's Freight Table

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bob's Auto Parts - Freight Costs</title>
  </head>
  <body>
```

```
<table style="border: 0px; padding: 3px">
<tr>
  <td style="background: #cccccc; text-align: center;">Distance</td>
  <td style="background: #cccccc; text-align: center;">Cost</td>
</tr>
<tr>
  <td style="text-align: right;">50</td>
  <td style="text-align: right;">5</td>
</tr>
<tr>
  <td style="text-align: right;">100</td>
  <td style="text-align: right;">10</td>
</tr>
<tr>
  <td style="text-align: right;">150</td>
  <td style="text-align: right;">15</td>
</tr>
<tr>
  <td style="text-align: right;">200</td>
  <td style="text-align: right;">20</td>
</tr>
<tr>
  <td style="text-align: right;">250</td>
  <td style="text-align: right;">25</td>
</tr>
</table>
</body>
</html>
```

Rather than requiring an easily bored human—who must be paid for his time—to type the HTML, having a cheap and tireless computer do it would be helpful.

Loop statements tell PHP to execute a statement or block repeatedly.

while Loops

The simplest kind of loop in PHP is the `while` loop. Like an `if` statement, it relies on a condition. The difference between a `while` loop and an `if` statement is that an `if` statement executes the code that follows it only once if the condition is `true`. A `while` loop executes the block repeatedly for as long as the condition is `true`.

You generally use a `while` loop when you don't know how many iterations will be required to make the condition true. If you require a fixed number of iterations, consider using a `for` loop.

The basic structure of a `while` loop is

```
while( condition ) expression;
```

The following `while` loop will display the numbers from 1 to 5:

```
$num = 1;
while ($num <= 5 ) {
    echo $num."<br />";
    $num++;
}
```

At the beginning of each iteration, the condition is tested. If the condition is `false`, the block will not be executed and the loop will end. The next statement after the loop will then be executed.

You can use a `while` loop to do something more useful, such as display the repetitive freight table in Figure 1.7. Listing 1.3 uses a `while` loop to generate the freight table.

Listing 1.3 **freight.php**—Generating Bob's Freight Table with PHP

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bob's Auto Parts - Freight Costs</title>
  </head>
  <body>
    <table style="border: 0px; padding: 3px">
      <tr>
        <td style="background: #cccccc; text-align: center;">Distance</td>
        <td style="background: #cccccc; text-align: center;">Cost</td>
      </tr>

      <?php
        $distance = 50;
        while ($distance <= 250) {
          echo "<tr>
            <td style='text-align: right;\">".$distance."</td>
            <td style='text-align: right;\">".($distance / 10)."</td>
          </tr>\n";
          $distance += 50;
        }
      ?>

    </table>
  </body>
</html>
```

To make the HTML generated by the script readable, you need to include newlines and spaces. As already mentioned, browsers ignore this whitespace, but it is important for human readers. You often need to look at the HTML if your output is not what you were seeking.

In Listing 1.3, you can see `\n` inside some of the strings. When inside a double-quoted string, this character sequence represents a newline character.

for and foreach Loops

The way that you used the `while` loops in the preceding section is very common. You set a counter to begin with. Before each iteration, you test the counter in a condition. And at the end of each iteration, you modify the counter.

You can write this style of loop in a more compact form by using a `for` loop. The basic structure of a `for` loop is

```
for( expression1; condition; expression2)
    expression3;
```

- *expression1* is executed once at the start. Here, you usually set the initial value of a counter.
- The *condition* expression is tested before each iteration. If the expression returns `false`, iteration stops. Here, you usually test the counter against a limit.
- *expression2* is executed at the end of each iteration. Here, you usually adjust the value of the counter.
- *expression3* is executed once per iteration. This expression is usually a block of code and contains the bulk of the loop code.

You can rewrite the `while` loop example in Listing 1.3 as a `for` loop. In this case, the PHP code becomes

```
<?php
for ($distance = 50; $distance <= 250; $distance += 50) {
    echo "<tr>
        <td style=\"text-align: right;\">\".$distance.\"</td>
        <td style=\"text-align: right;\">\".($distance / 10).\"</td>
    </tr>\n\"; }
?>
```

Both the `while` and `for` versions are functionally identical. The `for` loop is somewhat more compact, saving two lines.

Both these loop types are equivalent; neither is better or worse than the other. In a given situation, you can use whichever you find more intuitive.

As a side note, you can combine variable variables with a `for` loop to iterate through a series of repetitive form fields. If, for example, you have form fields with names such as `name1`, `name2`, `name3`, and so on, you can process them like this:

```
for ($i=1; $i <= $numnames; $i++){
    $temp= "name$i";
    echo htmlspecialchars($temp). '<br />'; // or whatever processing you want to do
}
```

By dynamically creating the names of the variables, you can access each of the fields in turn.

As well as the `for` loop, there is a `foreach` loop, designed specifically for use with arrays. We discuss how to use it in Chapter 3.

do...while Loops

The final loop type we describe behaves slightly differently. The general structure of a `do...while` statement is

```
do
    expression;
while( condition );
```

A `do...while` loop differs from a `while` loop because the condition is tested at the end. This means that in a `do...while` loop, the statement or block within the loop is always executed at least once.

Even if you consider this example in which the condition will be `false` at the start and can never become `true`, the loop will be executed once before checking the condition and ending:

```
$num = 100;
do{
    echo $num."<br />";
}while ( $num < 1 ) ;
```

Breaking Out of a Control Structure or Script

If you want to stop executing a piece of code, you can choose from three approaches, depending on the effect you are trying to achieve.

If you want to stop executing a loop, you can use the `break` statement as previously discussed in the section on `switch`. If you use the `break` statement in a loop, execution of the script will continue at the next line of the script after the loop.

If you want to jump to the next loop iteration, you can instead use the `continue` statement.

If you want to finish executing the entire PHP script, you can use `exit`. This approach is typically useful when you are performing error checking. For example, you could modify the earlier example as follows:

```
if($totalqty == 0){
    echo "You did not order anything on the previous page!<br />";
    exit;
}
```

The call to `exit` stops PHP from executing the remainder of the script.

Employing Alternative Control Structure Syntax

For all the control structures we have looked at, there is an alternative form of syntax. It consists of replacing the opening brace (`{}`) with a colon (`:`) and the closing brace with a new keyword, which will be `endif`, `endswitch`, `endwhile`, `endfor`, or `endforeach`, depending on which control structure is being used. No alternative syntax is available for `do...while` loops.

For example, the code

```
if ($totalqty == 0) {
    echo "You did not order anything on the previous page!<br />";
    exit;
}
```

could be converted to this alternative syntax using the keywords `if` and `endif`:

```
if ($totalqty == 0) :
    echo "You did not order anything on the previous page!<br />";
    exit;
endif;
```

Using declare

One other control structure in PHP, the `declare` structure, is not used as frequently in day-to-day coding as the other constructs. The general form of this control structure is as follows:

```
declare (directive)
{
    // block
}
```

This structure is used to set *execution directives* for the block of code—that is, rules about how the following code is to be run. Currently, only two execution directives, `ticks` and `encoding`, have been implemented.

You use `ticks` by inserting the directive `ticks=n`. It allows you to run a specific function every `n` lines of code inside the code block, which is principally useful for profiling and debugging.

The `encoding` directive is used to set encoding for a particular script, as follows:

```
declare(encoding='UTF-8');
```

In this case, the `declare` statement may not be followed by a code block if you are using namespaces. We'll talk about namespaces more later.

The `declare` control structure is mentioned here only for completeness. We consider some examples showing how to use `tick` functions in Chapters 25, “Using PHP and MySQL for Large Projects,” and 26, “Debugging and Logging.”

Next

Now you know how to receive and manipulate the customer’s order. In the next chapter, you’ll learn how to store the order so that it can be retrieved and fulfilled later.

Storing and Retrieving Data

Now that you know how to access and manipulate data entered in an HTML form, you can look at ways of storing that information for later use. In most cases, including the example from the previous chapter, you'll want to store this data and load it later. In this case, you need to write customer orders to storage so that they can be filled later.

In this chapter, you learn how to write the customer's order from the previous example to a file and read it back. You also learn why this isn't always a good solution. When you have large numbers of orders, you should use a database management system such as MySQL instead.

Key topics covered in this chapter include

- Saving data for later
- Opening a file
- Creating and writing to a file
- Closing a file
- Reading from a file
- Locking files
- Deleting files
- Using other useful file functions
- Doing it a better way: using database management systems

Saving Data for Later

You can store data in two basic ways: in flat files or in a database.

A flat file can have many formats, but in general, when we refer to a *flat file*, we mean a simple text file. For this chapter's example, you will write customer orders to a text file, one order per line.

Writing orders this way is very simple, but also limiting, as you'll see later in this chapter. If you're dealing with information of any reasonable volume, you'll probably want to use a database instead. However, flat files have their uses, and in some situations you need to know how to use them.

The processes of writing to and reading from files are similar in many programming languages. If you've done any C programming or Unix shell scripting, these procedures will seem familiar to you.

Storing and Retrieving Bob's Orders

In this chapter, you use a slightly modified version of the order form you looked at in the preceding chapter. Begin with this form and the PHP code you wrote to process the order data.

We've modified the form to include a quick way to obtain the customer's shipping address. You can see this modified form in Figure 2.1.

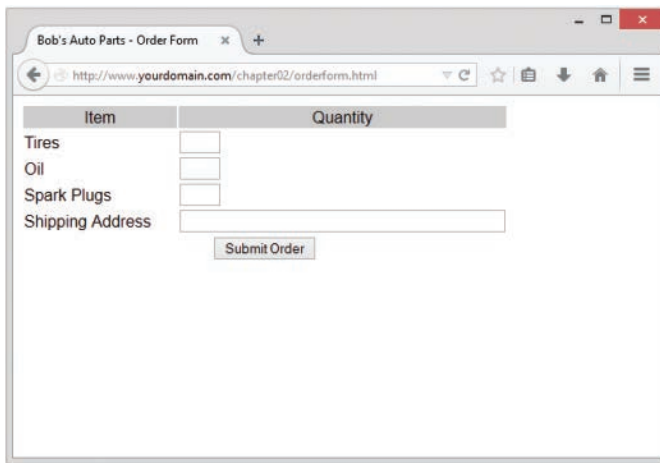
The image shows a web browser window with the title "Bob's Auto Parts - Order Form". The address bar shows "http://www.yourdomain.com/chapter02/orderform.html". The form itself has a table-like structure with two columns: "Item" and "Quantity". Under "Item", there are four rows: "Tires", "Oil", "Spark Plugs", and "Shipping Address". Each row has a corresponding empty input box in the "Quantity" column. Below the "Shipping Address" row, there is a "Submit Order" button.

Figure 2.1 This version of the order form gets the customer's shipping address

The form field for the shipping address is called `address`. This gives you a variable you can access as `$_REQUEST['address']` or `$_POST['address']` or `$_GET['address']`, depending on the form submission method. (See Chapter 1, "PHP Crash Course," for details.)

In this chapter, you write each order that comes in to the same file. Then you construct a web interface for Bob's staff to view the orders that have been received.

Processing Files

Writing data to a file requires three steps:

1. Open the file. If the file doesn't already exist, you need to create it.
2. Write the data to the file.
3. Close the file.

Similarly, reading data from a file takes three steps:

1. Open the file. If you cannot open the file (for example, if it doesn't exist), you need to recognize this and exit gracefully.
2. Read data from the file.
3. Close the file.

When you want to read data from a file, you have many choices about how much of the file to read at a time. We'll describe some common choices in detail. For now, we'll start at the beginning by opening a file.

Opening a File

To open a file in PHP, you use the `fopen()` function. When you open the file, you need to specify how you intend to use it. This is known as the *file mode*.

Choosing File Modes

The operating system on the server needs to know what you want to do with a file that you are opening. It needs to know whether the file can be opened by another script while you have it open and whether you (or the script owner) have permission to use it in the requested way. Essentially, file modes give the operating system a mechanism to determine how to handle access requests from other people or scripts and a method to check that you have access and permission to a particular file.

You need to make three choices when opening a file:

1. You might want to open a file for reading only, for writing only, or for both reading and writing.
2. If writing to a file, you might want to overwrite any existing contents of a file or append new data to the end of the file. You also might like to terminate your program gracefully instead of overwriting a file if the file already exists.
3. If you are trying to write to a file on a system that differentiates between binary and text files, you might need to specify this fact.

The `fopen()` function supports combinations of these three options.

Using `fopen()` to Open a File

Assume that you want to write a customer order to Bob's order file. You can open this file for writing with the following:

```
$fp = fopen("$document_root/./orders/orders.txt", 'w');
```

When `fopen()` is called, it expects two, three, or four parameters. Usually, you use two, as shown in this code line.

The first parameter should be the file you want to open. You can specify a path to this file, as in the preceding code; here, the `orders.txt` file is in the `orders` directory. We used the PHP built-in variable `$_SERVER['DOCUMENT_ROOT']` but, as with the cumbersome full names for form variables, we assigned a shorter name.

This variable points at the base of the document tree on your web server. This code line uses `..` to mean “the parent directory of the document root directory.” This directory is outside the document tree, for security reasons. In this case, we do not want this file to be web accessible except through the interface that we provide. This path is called a *relative path* because it describes a position in the file system relative to the document root.

As with the short names given form variables, you need the following line at the start of your script

```
$document_root = $_SERVER['DOCUMENT_ROOT'];
```

to copy the contents of the long-style variable to the short-style name.

You could also specify an *absolute path* to the file. This is the path from the root directory (`/` on a Unix system and typically `C:\` on a Windows system). On our Unix server, this path could be something like `/data/orders`. If no path is specified, the file will be created or looked for in the same directory as the script itself. The directory used will vary if you are running PHP through some kind of CGI wrapper and depends on your server configuration.

In a Unix environment, you use forward slashes (`/`) in directory paths. If you are using a Windows platform, you can use forward (`/`) or backslashes (`\`). If you use backslashes, they must be escaped (marked as a special character) for `fopen()` to understand them properly. To escape a character, you simply add an additional backslash in front of it, as shown in the following:

```
$fp = fopen("$document_root\\..\\orders\\orders.txt", 'w');
```

Very few people use backslashes in paths within PHP because it means the code will work only in Windows environments. If you use forward slashes, you can often move your code between Windows and Unix machines without alteration.

The second `fopen()` parameter is the file mode, which should be a string. This string specifies what you want to do with the file. In this case, we are passing `'w'` to `fopen()`; this means “open the file for writing.” A summary of file modes is shown in Table 2.1.

Table 2.1 Summary of File Modes for `fopen()`

Mode	Mode Name	Meaning
<code>r</code>	Read	Open the file for reading, beginning from the start of the file.
<code>r+</code>	Read	Open the file for reading and writing, beginning from the start of the file.
<code>w</code>	Write	Open the file for writing, beginning from the start of the file. If the file already exists, delete the existing contents. If it does not exist, try to create it.
<code>w+</code>	Write	Open the file for writing and reading, beginning from the start of the file. If the file already exists, delete the existing contents. If it does not exist, try to create it.
<code>x</code>	Cautious write	Open the file for writing, beginning from the start of the file. If the file already exists, it will not be opened, <code>fopen()</code> will return <code>false</code> , and PHP will generate a warning.
<code>x+</code>	Cautious write	Open the file for writing and reading, beginning from the start of the file. If the file already exists, it will not be opened, <code>fopen()</code> will return <code>false</code> , and PHP will generate a warning.
<code>a</code>	Append	Open the file for appending (writing) only, starting from the end of the existing contents, if any. If it does not exist, try to create it.
<code>a+</code>	Append	Open the file for appending (writing) and reading, starting from the end of the existing contents, if any. If it does not exist, try to create it.
<code>b</code>	Binary	Used in conjunction with one of the other modes. You might want to use this mode if your file system differentiates between binary and text files. Windows systems differentiate; Unix systems do not. The PHP developers recommend you always use this option for maximum portability. It is the default mode.
<code>t</code>	Text	Used in conjunction with one of the other modes. This mode is an option only in Windows systems. It is not recommended except before you have ported your code to work with the <code>b</code> option.

The right file mode to choose depends on how the system will be used. We used `'w'` in this example which allows only one order to be stored in the file. Each time a new order is taken, it overwrites the previous order. This usage is probably not very sensible, so you would be better off specifying append mode (and binary mode, as recommended):

```
$fp = fopen("$document_root/./orders/orders.txt", 'ab');
```

The third parameter of `fopen()` is optional. You can use it if you want to search the `include_path` (set in your PHP configuration; see Appendix A, “Installing Apache, PHP, and MySQL”) for a file. If you want to do this, set this parameter to `true`. If you tell PHP to search the `include_path`, you do not need to provide a directory name or path:

```
$fp = fopen('orders.txt', 'ab', true);
```

The fourth parameter is also optional. The `fopen()` function allows filenames to be prefixed with a protocol (such as `http://`) and opened at a remote location. Some protocols allow for an extra parameter. We look at this use of the `fopen()` function in the next section of this chapter.

If `fopen()` opens the file successfully, a resource that is effectively a handle or pointer to the file is returned and should be stored in a variable—in this case, `$fp`. You use this variable to access the file when you actually want to read from or write to it.

Opening Files Through FTP or HTTP

In addition to opening local files for reading and writing, you can open files via FTP, HTTP, and other protocols using `fopen()`. You can disable this capability by turning off the `allow_url_fopen` directive in the `php.ini` file. If you have trouble opening remote files with `fopen()`, check your `php.ini` file.

If the filename you use begins with `ftp://`, a passive mode FTP connection will be opened to the server you specify and a pointer to the start of the file will be returned.

If the filename you use begins with `http://`, an HTTP connection will be opened to the server you specify and a pointer to the response will be returned.

Remember that the domain names in your URL are not case sensitive, but the path and filename might be.

Addressing Problems Opening Files

An error you might make is trying to open a file you don't have permission to read from or write to. (This error occurs commonly on Unix-like operating systems, but you may also see it occasionally under Windows.) When you do, PHP gives you a warning similar to the one shown in Figure 2.2.

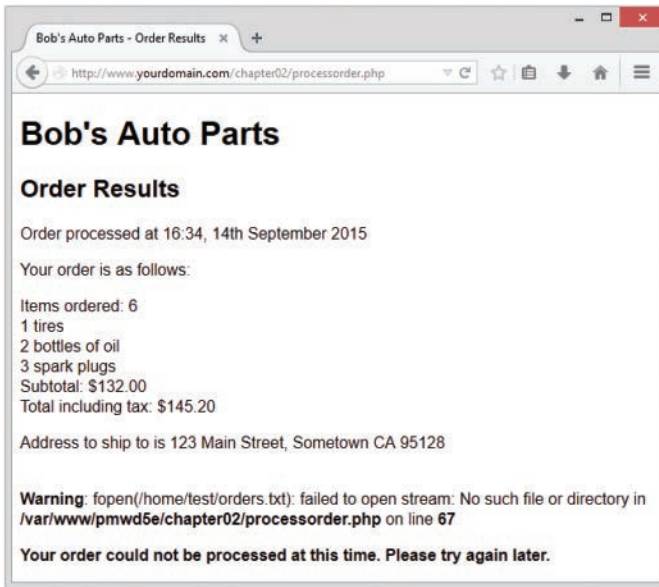


Figure 2.2 PHP specifically warns you when a file can't be opened

If you receive this error, you need to make sure that the user under which the script runs has permission to access the file you are trying to use. Depending on how your server is set up, the script might be running as the web server user or as the owner of the directory where the script is located.

On most systems, the script runs as the web server user. If your script is on a Unix system in the `~/public_html/chapter02/` directory, for example, you could create a group-writable directory in which to store the order by typing the following:

```
mkdir path/to/orders
chgrp apache path/to/orders
chmod 775 path/to/orders
```

You could also choose to change ownership of the file to the web server user. Some people will choose to make the file world-writable as a shortcut here, but bear in mind that directories and files that anybody can write to are dangerous. In particular, directories that are accessible directly from the Web should not be writable. For this reason, our `orders` directory is outside the document tree. We discuss security more in Chapter 15, “Building a Secure Web Application.”

Incorrect permission setting is probably the most common thing that can go wrong when opening a file, but it's not the only thing. If you can't open the file, you really need to know this so that you don't try to read data from or write data to it.

If the call to `fopen()` fails, the function will return `false`. It will also cause PHP to emit a `warning_level` error (`E_WARNING`). You can deal with the error in a more user-friendly way by suppressing PHP's error message and giving your own:

```
@$fp = fopen("$document_root/../../orders/orders.txt", 'ab');
if (!$fp){
    echo "<p><strong> Your order could not be processed at this time.  "
        .Please try again later.</strong></p></body></html>";
    exit;
}
```

The `@` symbol in front of the call to `fopen()` tells PHP to suppress any errors resulting from the function call. Usually, it's a good idea to know when things go wrong, but in this case we're going to deal with that problem elsewhere.

You can also write this line as follows:

```
$fp = @fopen("$document_root/../../orders/orders.txt", 'a');
```

Using this method tends to make it less obvious that you are using the error suppression operator, so it may make your code harder to debug.

Note

In general, use of the error suppression operator is not considered good style, so consider it a shortcut for now. The method described here is a simplistic way of dealing with errors. We look at a more elegant method for error handling in Chapter 7, "Error and Exception Handling." But one thing at a time.

The `if` statement tests the variable `$fp` to see whether a valid file pointer was returned from the `fopen()` call; if not, it prints an error message and ends script execution.

The output when using this approach is shown in Figure 2.3.

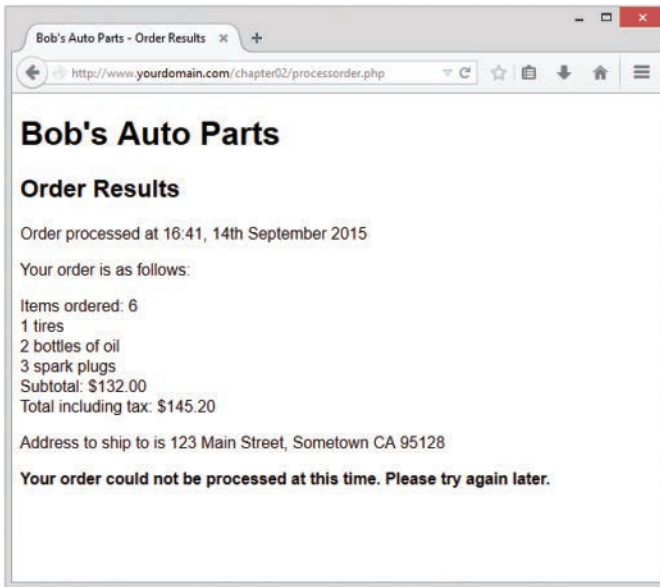


Figure 2.3 Using your own error messages instead of PHP's is more user friendly

Writing to a File

Writing to a file in PHP is relatively simple. You can use either of the functions `fwrite()` (file write) or `fputs()` (file put string); `fputs()` is an alias to `fwrite()`. You call `fwrite()` in the following way:

```
fwrite($fp, $outputstring);
```

This function call tells PHP to write the string stored in `$outputstring` to the file pointed to by `$fp`.

An alternative to `fwrite()` is the `file_put_contents()` function. It has the following prototype:

```
int file_put_contents ( string filename,
                      mixed data
                      [, int flags
                      [, resource context]])
```

This function writes the string contained in `data` to the file named in `filename` without any need for an `fopen()` (or `fclose()`) function call. This function is the half of a matched pair, the other half being `file_get_contents()`, which we discuss shortly. You most commonly use the `flags` and `context` optional parameters when writing to remote files using, for example, HTTP or FTP. (We discuss these functions in Chapter 18, "Using Network and Protocol Functions.")

Parameters for `fwrite()`

The function `fwrite()` actually takes three parameters, but the third one is optional. The prototype for `fwrite()` is

```
int fwrite ( resource handle, string [, int length])
```

The third parameter, *length*, is the maximum number of bytes to write. If this parameter is supplied, `fwrite()` will write *string* to the file pointed to by *handle* until it reaches the end of *string* or has written *length* bytes, whichever comes first.

You can obtain the string length by using PHP's built-in `strlen()` function, as follows:

```
fwrite($fp, $outputstring, strlen($outputstring));
```

You may want to use this third parameter when writing in binary mode because it helps avoid some cross-platform compatibility issues.

File Formats

When you are creating a data file like the one in the example, the format in which you store the data is completely up to you. (However, if you are planning to use the data file in another application, you may have to follow that application's rules.)

Now construct a string that represents one record in the data file. You can do this as follows:

```
$outputstring = $date."\t".$tireqty.' tires \t'.$oilqty.' oil\t'
                . $sparkqty.' spark plugs\t\${'.$totalamount
                . '\t'. $address.'\n';
```

In this simple example, you store each order record on a separate line in the file. Writing one record per line gives you a simple record separator in the newline character. Because newlines are invisible, you can represent them with the control sequence `"\n"`.

Throughout the book, we write the data fields in the same order every time and separate fields with a tab character. Again, because a tab character is invisible, it is represented by the control sequence `"\t"`. You may choose any sensible delimiter that is easy to read back.

The separator or delimiter character should be something that will certainly not occur in the input, or you should process the input to remove or escape out any instances of the delimiter. For now, if you look at the full code listing, you'll see that we have used a regular expression function (`preg_replace()`) to strip out potentially problematic characters. We will explain this fully when we look at processing input in Chapter 4, "String Manipulation and Regular Expressions."

Using a special field separator allows you to split the data back into separate variables more easily when you read the data back. We cover this topic in Chapter 3, "Using Arrays," and Chapter 4. Here, we treat each order as a single string.

After a few orders are processed, the contents of the file look something like the example shown in Listing 2.1.

Listing 2.1 **orders.txt**—Example of What the Orders File Might Contain

```
18:55, 16th April 2013  4 tires  1 oil  6 spark plugs  $477.4 22 Short St, Smalltown
18:56, 16th April 2013  1 tires  0 oil  0 spark plugs  $110  33 Main Rd, Oldtown
18:57, 16th April 2013  0 tires  1 oil  4 spark plugs  $28.6 127 Acacia St,
Springfield
```

Closing a File

After you’ve finished using a file, you need to close it. You should do this by using the `fclose()` function as follows:

```
fclose($fp);
```

This function returns `true` if the file was successfully closed or `false` if it wasn’t. This process is much less likely to go wrong than opening a file in the first place, so in this case we’ve chosen not to test it.

The complete listing for the final version of `processorder.php` is shown in Listing 2.2.

Listing 2.2 **processorder.php**—Final Version of the Order Processing Script

```
<?php
    // create short variable names
    $tireqty = (int) $_POST['tireqty'];
    $oilqty = (int) $_POST['oilqty'];
    $sparkqty = (int) $_POST['sparkqty'];
    $address = preg_replace('/\t|\R/', ' ', $_POST['address']);
    $document_root = $_SERVER['DOCUMENT_ROOT'];
    $date = date('H:i, jS F Y');
?>
<!DOCTYPE html>
<html>
    <head>
        <title>Bob's Auto Parts - Order Results</title>
    </head>
    <body>
        <h1>Bob's Auto Parts</h1>
        <h2>Order Results</h2>
        <?php
            echo "<p>Order processed at ".date('H:i, jS F Y')."</p>";
            echo "<p>Your order is as follows: </p>";

            $totalqty = 0;
            $totalamount = 0.00;

            define('TIREPRICE', 100);
```

```

define('OILPRICE', 10);
define('SPARKPRICE', 4);

$totalqty = $tireqty + $oilqty + $sparkqty;
echo "<p>Items ordered: ".$totalqty."<br />";

if ($totalqty == 0) {
    echo "You did not order anything on the previous page!<br />";
} else {
    if ($tireqty > 0) {
        echo htmlspecialchars($tireqty).' tires<br />';
    }
    if ($oilqty > 0) {
        echo htmlspecialchars($oilqty).' bottles of oil<br />';
    }
    if ($sparkqty > 0) {
        echo htmlspecialchars($sparkqty).' spark plugs<br />';
    }
}

$totalamount = $tireqty * TIREPRICE
               + $oilqty * OILPRICE
               + $sparkqty * SPARKPRICE;

echo "Subtotal: ".$number_format($totalamount,2)."<br />";

$taxrate = 0.10; // local sales tax is 10%
$totalamount = $totalamount * (1 + $taxrate);
echo "Total including tax: ".$number_format($totalamount,2)."</p>";

echo "<p>Address to ship to is ".htmlspecialchars($address)."</p>";

$outputstring = $date."\t".$tireqty." tires \t".$oilqty." oil\t"
               . $sparkqty." spark plugs\t".$totalamount
               . "\t". $address."\n";

// open file for appending
@$fp = fopen("$document_root/../orders/orders.txt", 'ab');

if (!$fp) {
    echo "<p><strong> Your order could not be processed at this time.
        Please try again later.</strong></p>";
    exit;
}

flock($fp, LOCK_EX);

```

```

        fwrite($fp, $outputstring, strlen($outputstring));
        flock($fp, LOCK_UN);
        fclose($fp);

        echo "<p>Order written.</p>";
    ?>
</body>
</html>

```

Reading from a File

Right now, Bob's customers can leave their orders via the Web, but if Bob's staff members want to look at the orders, they have to open the files themselves.

Let's create a web interface to let Bob's staff read the files easily. The code for this interface is shown in Listing 2.3.

Listing 2.3 **vieworders.php**—Staff Interface to the Orders File

```

<?php
    // create short variable name
    $document_root = $_SERVER['DOCUMENT_ROOT'];
?>
<!DOCTYPE html>
<html>
    <head>
        <title>Bob's Auto Parts - Order Results</title>
    </head>
    <body>
        <h1>Bob's Auto Parts</h1>
        <h2>Customer Orders</h2>
        <?php
            @$fp = fopen("$document_root/../orders/orders.txt", 'rb');
            flock($fp, LOCK_SH); // lock file for reading

            if (! $fp) {
                echo "<p><strong>No orders pending.<br />
                    Please try again later.</strong></p>";
                exit;
            }

            while (!feof($fp)) {
                $order= fgets($fp);
                echo htmlspecialchars($order)."<br />";
            }

```

```

        flock($fp, LOCK_UN); // release read lock
        fclose($fp);
    ?>
</body>
</html>

```

This script follows the sequence we described earlier: open the file, read from the file, close the file. The output from this script using the data file from Listing 2.1 is shown in Figure 2.4.

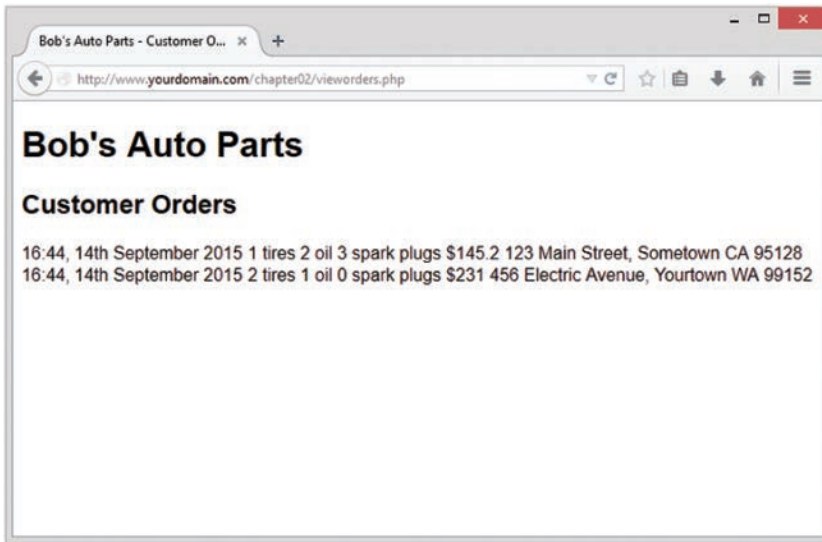


Figure 2.4 The `vieworders.php` script displays all the orders currently in the `orders.txt` file in the browser window

Let's look at the functions in this script in detail.

Opening a File for Reading: `fopen()`

Again, you open the file by using `fopen()`. In this case, you open the file for reading only, so you use the file mode `'rb'`:

```
$fp = fopen("$document_root/../../orders/orders.txt", 'rb');
```

Knowing When to Stop: `feof()`

In this example, you use a `while` loop to read from the file until the end of the file is reached. The `while` loop tests for the end of the file using the `feof()` function:

```
while (!feof($fp))
```

The `feof()` function takes a file handle as its single parameter. It returns `true` if the file pointer is at the end of the file. Although the name might seem strange, you can remember it easily if you know that `feof` stands for File End Of File.

In this case (and generally when reading from a file), you read from the file until EOF is reached.

Reading a Line at a Time: `fgets()`, `fgetss()`, and `fgetcsv()`

In this example, you use the `fgets()` function to read from the file:

```
$order= fgets($fp);
```

This function reads one line at a time from a file. In this case, it reads until it encounters a newline character (`\n`) or EOF.

You can use many different functions to read from files. The `fgets()` function, for example, is useful when you're dealing with files that contain plain text that you want to deal with in chunks.

An interesting variation on `fgets()` is `fgetss()`, which has the following prototype:

```
string fgetss(resource fp[, int length[, string allowable_tags]]);
```

This function is similar to `fgets()` except that it strips out any PHP and HTML tags found in the string. If you want to leave in any particular tags, you can include them in the `allowable_tags` string. You would use `fgetss()` for safety when reading a file written by somebody else or one containing user input. Allowing unrestricted HTML code in the file could mess up your carefully planned formatting. Allowing unrestricted PHP or JavaScript could give a malicious user an opportunity to create a security problem.

The function `fgetcsv()` is another variation on `fgets()`. It has the following prototype:

```
array fgetcsv ( resource fp, int length [, string delimiter
                [, string enclosure
                [, string escape]])
```

This function breaks up lines of files when you have used a delimiting character, such as the tab character (as we suggested earlier) or a comma (as commonly used by spreadsheets and other applications). If you want to reconstruct the variables from the order separately rather than as a line of text, `fgetcsv()` allows you to do this simply. You call it in much the same way as you would call `fgets()`, but you pass it the delimiter you used to separate fields. For example,

```
$order = fgetcsv($fp, 0, "\t");
```

This code would retrieve a line from the file and break it up wherever a tab (`\t`) was encountered. The results are returned in an array (`$order` in this code example). We cover arrays in more detail in Chapter 3.

The *length* parameter should be greater than the length in characters of the longest line in the file you are trying to read, or 0 if you do not want to limit the line length.

The *enclosure* parameter specifies what each field in a line is surrounded by. If not specified, it defaults to " (a double quotation mark).

Reading the Whole File: `readfile()`, `fpassthru()`, `file()`, and `file_get_contents()`

Instead of reading from a file a line at a time, you can read the whole file in one go. Here are four different ways you can do this.

The first uses `readfile()`. You can replace almost the entire script you wrote previously with one line:

```
readfile("$document_root/../../orders/orders.txt");
```

A call to the `readfile()` function opens the file, echoes the content to standard output (the browser), and then closes the file. The prototype for `readfile()` is

```
int readfile(string filename, [bool use_include_path[, resource context]] );
```

The optional second parameter specifies whether PHP should look for the file in the `include_path` and operates the same way as in `fopen()`. The optional `context` parameter is used only when files are opened remotely via, for example, HTTP; we cover such usage in more detail in Chapter 18. The function returns the total number of bytes read from the file.

Second, you can use `fpassthru()`. To do so, you need to open the file using `fopen()` first. You can then pass the file pointer as an argument to `fpassthru()`, which dumps the contents of the file from the pointer's position onward to standard output. It closes the file when it is finished.

You can use `fpassthru()` as follows:

```
$fp = fopen("$document_root/../../orders/orders.txt", 'rb');
fpassthru($fp);
```

The function `fpassthru()` returns `true` if the read is successful and `false` otherwise.

The third option for reading the whole file is using the `file()` function. This function is identical to `readfile()` except that instead of echoing the file to standard output, it turns it into an array. We cover this function in more detail when we look at arrays in Chapter 3. Just for reference, you would call it using

```
$filearray = file("$document_root/../../orders/orders.txt");
```

This line reads the entire file into the array called `$filearray`. Each line of the file is stored in a separate element of the array. Note that this function was not binary safe in older versions of PHP.

The fourth option is to use the `file_get_contents()` function. This function is identical to `readfile()` except that it returns the content of the file as a string instead of outputting it to the browser.

Reading a Character: `fgetc()`

Another option for file processing is to read a single character at a time from a file. You can do this by using the `fgetc()` function. It takes a file pointer as its only parameter and returns the next character in the file. You can replace the `while` loop in the original script with one that uses `fgetc()`, as follows:

```
while (!feof($fp)) {  
    $char = fgetc($fp);  
    if (!feof($fp))  
        echo ($char=="\n" ? "<br />": $char);  
}
```

This code reads a single character at a time from the file using `fgetc()` and stores it in `$char`, until the end of the file is reached. It then does a little processing to replace the text end-of-line characters (`\n`) with HTML line breaks (`
`).

This is just to clean up the formatting. If you try to output the file with newlines between records, the whole file will be printed on a single line. (Try it and see.) Web browsers do not render whitespace, such as newlines, so you need to replace them with HTML linebreaks (`
`) instead. You can use the ternary operator to do this neatly.

A minor side effect of using `fgetc()` instead of `fgets()` is that `fgetc()` returns the EOF character, whereas `fgets()` does not. You need to test `feof()` again after you've read the character because you don't want to echo the EOF to the browser.

Reading a file character by character is not generally sensible or efficient unless for some reason you actually want to process it character by character.

Reading an Arbitrary Length: `fread()`

The final way you can read from a file is to use the `fread()` function to read an arbitrary number of bytes from the file. This function has the following prototype:

```
string fread(resource fp, int length);
```

It reads up to *length* bytes, to the end of the file or network packet, whichever comes first.

Using Other File Functions

Numerous other file functions are useful from time to time. Some that we have found handy are described next.

Checking Whether a File Is There: `file_exists()`

If you want to check whether a file exists without actually opening it, you can use `file_exists()`, as follows:

```
if (file_exists("$document_root/../orders/orders.txt")) {
    echo 'There are orders waiting to be processed.';
} else {
    echo 'There are currently no orders.';
}
```

Determining How Big a File Is: `filesize()`

You can check the size of a file by using the `filesize()` function:

```
echo filesize("$document_root/../orders/orders.txt");
```

It returns the size of a file in bytes and can be used in conjunction with `fread()` to read a whole file (or some fraction of the file) at a time. You can even replace the entire original script with the following:

```
$fp = fopen("$document_root/../orders/orders.txt", 'rb');
echo nl2br(fread( $fp, filesize("$document_root/../orders/orders.txt")));
fclose( $fp );
```

The `nl2br()` function converts the `\n` characters in the output to HTML line breaks (`
`).

Deleting a File: `unlink()`

If you want to delete the order file after the orders have been processed, you can do so by using `unlink()`. (There is no function called `delete`.) For example,

```
unlink("$document_root/../orders/orders.txt");
```

This function returns `false` if the file could not be deleted. This situation typically occurs if the permissions on the file are insufficient or if the file does not exist.

Navigating Inside a File: `rewind()`, `fseek()`, and `ftell()`

You can manipulate and discover the position of the file pointer inside a file by using `rewind()`, `fseek()`, and `ftell()`.

The `rewind()` function resets the file pointer to the beginning of the file. The `ftell()` function reports how far into the file the pointer is in bytes. For example, you can add the following lines to the bottom of the original script (before the `fclose()` command):

```
echo 'Final position of the file pointer is ' . (ftell($fp));
echo '<br />';
rewind($fp);
echo 'After rewind, the position is ' . (ftell($fp));
echo '<br />';
```

The output in the browser should be similar to that shown in Figure 2.5.

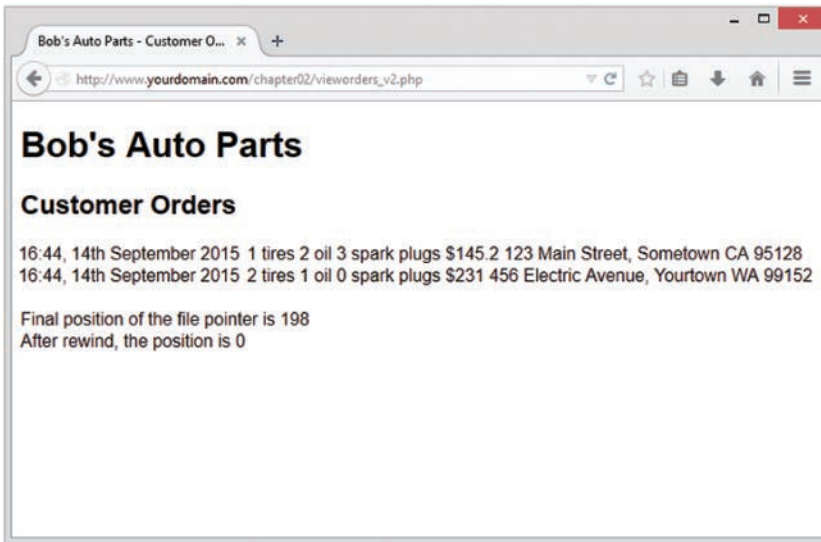


Figure 2.5 After reading the orders, the file pointer points to the end of the file, an offset of 198 bytes. The call to rewind sets it back to position 0, the start of the file

You can use the function `fseek()` to set the file pointer to some point within the file. Its prototype is

```
int fseek ( resource fp, int offset [, int whence])
```

A call to `fseek()` sets the file pointer `fp` at a point starting from `whence` and moving `offset` bytes into the file. The optional `whence` parameter defaults to the value `SEEK_SET`, which is effectively the start of the file. The other possible values are `SEEK_CUR` (the current location of the file pointer) and `SEEK_END` (the end of the file).

The `rewind()` function is equivalent to calling the `fseek()` function with an offset of zero. For example, you can use `fseek()` to find the middle record in a file or to perform a binary search. If you reach the level of complexity in a data file where you need to do these kinds of things, your life will be much easier if you use a built-for-purpose database.

Locking Files

Imagine a situation in which two customers are trying to order a product at the same time. (This situation is not uncommon, especially when your website starts to get any kind of traffic volume.) What if one customer calls `fopen()` and begins writing, and then the other customer calls `fopen()` and also begins writing? What will be the final contents of the file? Will it be the first order followed by the second order, or vice versa? Will it be one order or the other?

Or will it be something less useful, such as the two orders interleaved somehow? The answer depends on your operating system but is often impossible to know.

To avoid problems like this, you can use file locking. You use this feature in PHP by using the `flock()` function. This function should be called after a file has been opened but before any data is read from or written to the file.

The prototype for `flock()` is

```
bool flock (resource fp, int operation [, int &wouldblock])
```

You need to pass it a pointer to an open file and a constant representing the kind of lock you require. It returns true if the lock was successfully acquired and false if it was not. The optional third parameter will contain the value true if acquiring the lock would cause the current process to block (that is, have to wait).

The possible values for *operation* are shown in Table 2.2.

Table 2.2 **flock()** Operation Values

Value of Operation	Meaning
LOCK_SH	Reading lock. The file can be shared with other readers.
LOCK_EX	Writing lock. This operation is exclusive; the file cannot be shared.
LOCK_UN	The existing lock is released.
LOCK_NB	Blocking is prevented while you are trying to acquire a lock. (Not supported on Windows.)

If you are going to use `flock()`, you need to add it to all the scripts that use the file; otherwise, it is worthless.

Note that `flock()` does not work with NFS or other networked file systems. It also does not work with antique file systems that do not support locking, such as FAT. On some operating systems, it is implemented at the process level and does not work correctly if you are using a multithreaded server API.

To use it with the order example, you can alter `processorder.php` as follows:

```
@ $fp = fopen("$document_root/./orders/orders.txt", 'ab');

flock($fp, LOCK_EX);

if (!$fp) {
    echo "<p><strong> Your order could not be processed at this time.
        Please try again later.</strong></p></body></html>";
    exit;
}

fwrite($fp, $outputstring, strlen($outputstring));
```

```
flock($fp, LOCK_UN);  
fclose($fp);
```

You should also add locks to `vieworders.php`:

```
@$fp = fopen("$document_root/../orders/orders.txt", 'rb');  
flock($fp, LOCK_SH); // lock file for reading  
// read from file  
flock($fp, LOCK_UN); // release read lock  
fclose($fp);
```

The code is now more robust but still not perfect. What if two scripts tried to acquire a lock at the same time? This would result in a race condition, in which the processes compete for locks but it is uncertain which will succeed. Such a condition could cause more problems. You can do better by using a database.

A Better Way: Databases

So far, all the examples we have looked at use flat files. In Part II of this book, we look at how to use MySQL, a relational database management system (RDBMS), instead. You might ask, “Why would I bother?”

Problems with Using Flat Files

There are a number of problems in working with flat files:

- When a file grows large, working with it can be very slow.
- Searching for a particular record or group of records in a flat file is difficult. If the records are in order, you can use some kind of binary search in conjunction with a fixed-width record to search on a key field. If you want to find patterns of information (for example, you want to find all the customers who live in Sometown), you would have to read in each record and check it individually.
- Dealing with concurrent access can become problematic. You have seen how to lock files, but locking can cause the race condition we discussed earlier. It can also cause a bottleneck. With enough traffic on a site, a large group of users may be waiting for the file to be unlocked before they can place their order. If the wait is too long, people will go elsewhere to buy.
- All the file processing you have seen so far deals with a file using sequential processing; that is, you start from the beginning of the file and read through to the end. Inserting records into or deleting records from the middle of the file (random access) can be difficult because you end up reading the whole file into memory, making the changes, and writing the whole file out again. With a large data file, having to go through all these steps becomes a significant overhead.
- Beyond the limits offered by file permissions, there is no easy way of enforcing different levels of access to data.

How RDBMSs Solve These Problems

Relational database management systems address all these issues:

- RDBMSs can provide much faster access to data than flat files. And MySQL, the database system we use in this book, has some of the fastest benchmarks of any RDBMS.
- RDBMSs can be easily queried to extract sets of data that fit certain criteria.
- RDBMSs have built-in mechanisms for dealing with concurrent access so that you, as a programmer, don't have to worry about it.
- RDBMSs provide random access to your data.
- RDBMSs have built-in privilege systems. MySQL has particular strengths in this area.

Probably the main reason for using an RDBMS is that all (or at least most) of the functionality that you want in a data storage system has already been implemented. Sure, you could write your own library of PHP functions, but why reinvent the wheel?

In Part II of this book, “Using MySQL,” we discuss how relational databases work generally, and specifically how you can set up and use MySQL to create database-backed websites.

If you are building a simple system and don't feel you need a full-featured database but want to avoid the locking and other issues associated with using a flat file, you may want to consider using PHP's SQLite extension. This extension provides essentially an SQL interface to a flat file. In this book, we focus on using MySQL, but if you would like more information about SQLite, you can find it at <http://sqlite.org/> and <http://www.php.net/sqlite>.

Further Reading

For more information on interacting with the file system, you can go straight to Chapter 17, “Interacting with the File System and the Server.” In that part of the book, we talk about how to change permissions, ownership, and names of files; how to work with directories; and how to interact with the file system environment.

You may also want to read through the file system section of the PHP online manual at <http://www.php.net/filesystem>.

Next

In the next chapter, you learn what arrays are and how they can be used for processing data in your PHP scripts.

Using Arrays

This chapter shows you how to use an important programming construct: arrays. The variables used in the previous chapters were *scalar* variables, which store a single value. An *array* is a variable that stores a set or sequence of values. One array can have many elements, and each element can hold a single value, such as text or numbers, or another array. An array containing other arrays is known as a *multidimensional array*.

PHP supports arrays with both numerical and string indexes. You are probably familiar with numerically indexed arrays if you've used any other programming language, but you might not have seen arrays using string indexes before, although you may have seen similar things called hashes, maps, or dictionaries elsewhere. Rather than each element having a numeric index, you can use words or other meaningful information.

In this chapter, you continue developing the Bob's Auto Parts example using arrays to work more easily with repetitive information such as customer orders. Likewise, you write shorter, tidier code to do some of the things you did with files in the preceding chapter.

Key topics covered in this chapter include

- Numerically indexed arrays
- Non-numerically indexed arrays
- Array operators
- Multidimensional arrays
- Array sorting
- Array functions

What Is an Array?

You learned about scalar variables in Chapter 1, “PHP Crash Course.” A scalar variable is a named location in which to store a value; similarly, an array is a named place to store a *set* of values, thereby allowing you to group variables.

Bob's product list is the array for the example used in this chapter. In Figure 3.1, you can see a list of three products stored in an array format. These three products are stored in a single variable called `$products`. (We describe how to create a variable like this shortly.)

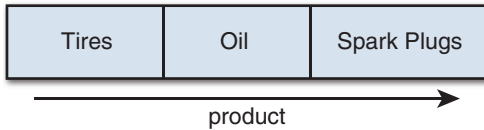


Figure 3.1 Bob's products can be stored in an array

After you have the information as an array, you can do a number of useful things with it. Using the looping constructs from Chapter 1, you can save work by performing the same actions on each value in the array. The whole set of information can be moved around as a single unit. This way, with a single line of code, all the values in the array can be passed to a function. For example, you might want to sort the products alphabetically. To achieve this, you could pass the entire array to PHP's `sort()` function.

The values stored in an array are called the array *elements*. Each array element has an associated *index* (also called a *key*) that is used to access the element. Arrays in many programming languages have numerical indices that typically start from zero or one.

PHP allows you to interchangeably use numbers or strings as the array indices. You can use arrays in the traditional numerically indexed way or set the keys to be whatever you like to make the indexing more meaningful and useful. (This approach may be familiar to you if you have used associative arrays, maps, hashes, or dictionaries in other programming languages.) The programming approach may vary a little depending on whether you are using standard numerically indexed arrays or more interesting index values.

We begin by looking at numerically indexed arrays and then move on to using user-defined keys.

Numerically Indexed Arrays

Numerically indexed arrays are supported in most programming languages. In PHP, the indices start at zero by default, although you can alter this value.

Initializing Numerically Indexed Arrays

To create the array shown in Figure 3.1, use the following line of PHP code:

```
$products = array( 'Tires', 'Oil', 'Spark Plugs' );
```

This code creates an array called `$products` containing the three values given: 'Tires', 'Oil', and 'Spark Plugs'. Note that, like `echo`, `array()` is actually a language construct rather than a function.

Since PHP 5.4, you can use a new shorthand syntax for creating arrays. This uses the `[` and `]` characters in place of the `array()` operator. For example, to create the array shown in Figure 3.1 with the shorthand syntax, you would use the following line of code:

```
$products = ['Tires', 'Oil', 'Spark Plugs'];
```

Depending on the contents you need in your array, you might not need to manually initialize them as in the preceding example. If you have the data you need in another array, you can simply copy one array to another using the `=` operator.

If you want an ascending sequence of numbers stored in an array, you can use the `range()` function to automatically create the array for you. The following statement creates an array called `numbers` with elements ranging from 1 to 10:

```
$numbers = range(1,10);
```

The `range()` function has an optional third parameter that allows you to set the step size between values. For instance, if you want an array of the odd numbers between 1 and 10, you could create it as follows:

```
$odds = range(1, 10, 2);
```

The `range()` function can also be used with characters, as in this example:

```
$letters = range('a', 'z');
```

If you have information stored in a file on disk, you can load the array contents directly from the file. We look at this topic later in this chapter under the heading “Loading Arrays from Files.”

If you have the data for your array stored in a database, you can load the array contents directly from the database. This process is covered in Chapter 11, “Accessing Your MySQL Database from the Web with PHP.”

You can also use various functions to extract part of an array or to reorder an array. We look at some of these functions later in this chapter under the heading “Performing Other Array Manipulations.”

Accessing Array Contents

To access the contents of a variable, you use its name. If the variable is an array, you access the contents using both the variable name and a key or index. The key or index indicates which of the values in the array you access. The index is placed in square brackets after the name. In other words, you can use `$products[0]`, `$products[1]`, and `$products[2]` to access each of the contents of the `$products` array.

You may also use the `{}` characters to access array elements instead of the `[]` characters if you prefer. For example, you could use `$products{0}` to access the first element of the `products` array.

By default, element zero is the first element in the array. The same numbering scheme is used in C, C++, Java, and a number of other languages, but it might take some getting used to if you are not familiar with it.

As with other variables, you change array elements' contents by using the `=` operator.

The following line replaces the first element in the array, `'Tires'`, with `'Fuses'`:

```
$products[0] = 'Fuses';
```

You can use the following line to add a new element—`'Fuses'`—to the end of the array, giving a total of four elements:

```
$products[3] = 'Fuses';
```

To display the contents, you could type this line:

```
echo "$products[0] $products[1] $products[2] $products[3]";
```

Note that although PHP's string parsing is pretty clever, you can confuse it. If you are having trouble with array or other variables not being interpreted correctly when embedded in a double-quoted string, you can either put them outside quotes or use complex syntax, which we discuss in Chapter 4, "String Manipulation and Regular Expressions." The preceding `echo` statement works correctly, but in many of the more complex examples later in this chapter, you will notice that the variables are outside the quoted strings.

Like other PHP variables, arrays do not need to be initialized or created in advance. They are automatically created the first time you use them.

The following code creates the same `$products` array created previously with the `array()` statement:

```
$products[0] = 'Tires';
$products[1] = 'Oil';
$products[2] = 'Spark Plugs';
```

If `$products` does not already exist, the first line will create a new array with just one element. The subsequent lines add values to the array. The array is dynamically resized as you add elements to it. This resizing capability is not present in many other programming languages.

Using Loops to Access the Array

Because the array is indexed by a sequence of numbers, you can use a `for` loop to more easily display its contents:

```
for ($i = 0; $i < 3; $i++) {
    echo $products[$i] . " ";
}
```

This loop provides similar output to the preceding code but requires less typing than manually writing code to work with each element in a large array. The ability to use a simple loop to access each element is a nice feature of arrays. You can also use the `foreach` loop, specially designed for use with arrays. In this example, you could use it as follows:

```
foreach ($products as $current) {  
    echo $current." ";  
}
```

This code stores each element in turn in the variable `$current` and prints it out.

Arrays with Different Indices

In the `$products` array, you allowed PHP to give each item the default index. This meant that the first item you added became item 0; the second, item 1; and so on. PHP also supports arrays in which you can associate any scalar key or index you want with each value.

Initializing an Array

The following code creates an array with product names as keys and prices as values:

```
$prices = array('Tires'=>100, 'Oil'=>10, 'Spark Plugs'=>4);
```

The symbol between the keys and values (`=>`) is simply an equal sign immediately followed by a greater than symbol.

Accessing the Array Elements

Again, you access the contents using the variable name and a key, so you can access the information stored in the `prices` array as `$prices['Tires']`, `$prices['Oil']`, and `$prices['Spark Plugs']`.

The following code creates the same `$prices` array. Instead of creating an array with three elements, this version creates an array with only one element and then adds two more:

```
$prices = array('Tires'=>100);  
$prices['Oil'] = 10;  
$prices['Spark Plugs'] = 4;
```

Here is another slightly different but equivalent piece of code. In this version, you do not explicitly create an array at all. The array is created for you when you add the first element to it:

```
$prices['Tires'] = 100;  
$prices['Oil'] = 10;  
$prices['Spark Plugs'] = 4;
```

Using Loops

Because the indices in an array are not numbers, you cannot use a simple counter in a `for` loop to work with the array. However, you can use the `foreach` loop or the `list()` and `each()` constructs.

The `foreach` loop has a slightly different structure when using non-numerically indexed arrays. You can use it exactly as you did in the previous example, or you can incorporate the keys as well:

```
foreach ($prices as $key => $value) {  
    echo $key." - ".$value."<br />";  
}
```

The following code lists the contents of the `$prices` array using the `each()` construct:

```
while ($element = each($prices)) {  
    echo $element['key']." - ".$element['value'];  
    echo "<br />";  
}
```

The output of this script fragment is shown in Figure 3.2.

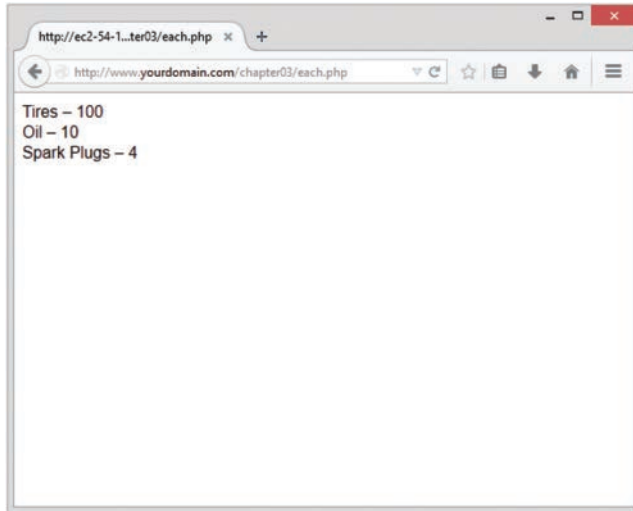


Figure 3.2 An `each()` statement can be used to loop through arrays

In Chapter 1, we looked at `while` loops and the `echo` statement. The preceding code uses the `each()` function, which we have not yet covered. This function returns the current element in an array and makes the next element the current one. Because we are calling `each()` within a `while` loop, it returns every element in the array in turn and stops when the end of the array is reached.

In this code, the variable `$element` is an array. When you call `each()`, it gives you an array with four values and the four indices to the array locations. The locations `key` and `0` contain the key of the current element, and the locations `value` and `1` contain the value of the current element. Although the one you choose makes no difference, we chose to use the named locations rather than the numbered ones.

There is a more elegant and common way of doing the same thing. The construct `list()` can be used to split an array into a number of values. You can separate each set of values that the `each()` function gives you like this:

```
while (list($product, $price) = each($prices)) {
    echo $product." - ".$price."<br />";
}
```

This line uses `each()` to take the current element from `$prices`, return it as an array, and make the next element current. It also uses `list()` to turn the 0 and 1 elements from the array returned by `each()` into two new variables called `$product` and `$price`.

When you are using `each()`, note that the array keeps track of the current element. If you want to use the array twice in the same script, you need to set the current element back to the start of the array using the function `reset()`. To loop through the `prices` array again, you type the following:

```
reset($prices);
while (list($product, $price) = each($prices)) {
    echo $product." - ".$price."<br />";
}
```

This code sets the current element back to the start of the array and allows you to go through again.

Array Operators

One set of special operators applies only to arrays. Most of them have an analogue in the scalar operators, as you can see by looking at Table 3.1.

Table 3.1 PHP's Array Operators

Operator	Name	Example	Result
+	Union	<code>\$a + \$b</code>	Union of <code>\$a</code> and <code>\$b</code> . The array <code>\$b</code> is appended to <code>\$a</code> , but any key clashes are not added.
==	Equality	<code>\$a == \$b</code>	True if <code>\$a</code> and <code>\$b</code> contain the same elements.
===	Identity	<code>\$a === \$b</code>	True if <code>\$a</code> and <code>\$b</code> contain the same elements, with the same types, in the same order.
!=	Inequality	<code>\$a != \$b</code>	True if <code>\$a</code> and <code>\$b</code> do not contain the same elements.
<>	Inequality	<code>\$a <> \$b</code>	Same as <code>!=</code> .
!==	Non-identity	<code>\$a !== \$b</code>	True if <code>\$a</code> and <code>\$b</code> do not contain the same elements, with the same types, in the same order.

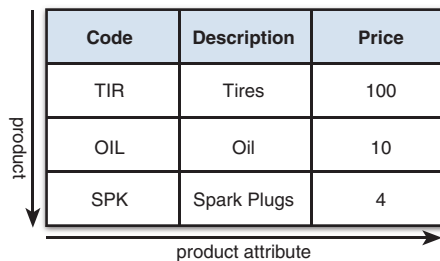
These operators are mostly fairly self-evident, but union requires some further explanation. The union operator tries to add the elements of `$b` to the end of `$a`. If elements in `$b` have the same keys as some elements already in `$a`, they will not be added. That is, no elements of `$a` will be overwritten.

You will notice that the array operators in Table 3.1 all have equivalent operators that work on scalar variables. As long as you remember that `+` performs addition on scalar types and union on arrays—even if you have no interest in the set arithmetic behind that behavior—the behaviors should make sense. You cannot usefully compare arrays to scalar types.

Multidimensional Arrays

Arrays do not have to be a simple list of keys and values; each location in the array can hold another array. This way, you can create a two-dimensional array. You can think of a two-dimensional array as a matrix, or grid, with width and height or rows and columns.

If you want to store more than one piece of data about each of Bob's products, you could use a two-dimensional array. Figure 3.3 shows Bob's products represented as a two-dimensional array with each row representing an individual product and each column representing a stored product attribute.



Code	Description	Price
TIR	Tires	100
OIL	Oil	10
SPK	Spark Plugs	4

Figure 3.3 You can store more information about Bob's products in a two-dimensional array

Using PHP, you would write the following code to set up the data in the array shown in Figure 3.3:

```
$products = array( array('TIR', 'Tires', 100 ),
                  array('OIL', 'Oil', 10 ),
                  array('SPK', 'Spark Plugs', 4 ) );
```

You can see from this definition that the `$products` array now contains three arrays.

To access the data in a one-dimensional array, recall that you need the name of the array and the index of the element. A two-dimensional array is similar, except that each element has two indices: a row and a column. (The top row is row 0, and the far-left column is column 0.)

To display the contents of this array, you could manually access each element in order like this:

```
echo '|'.$products[0][0].'|'.$products[0][1].'|'.$products[0][2].'|<br />';
echo '|'.$products[1][0].'|'.$products[1][1].'|'.$products[1][2].'|<br />';
echo '|'.$products[2][0].'|'.$products[2][1].'|'.$products[2][2].'|<br />';
```

Alternatively, you could place a `for` loop inside another `for` loop to achieve the same result:

```
for ($row = 0; $row < 3; $row++) {
    for ($column = 0; $column < 3; $column++) {
        echo '|'.$products[$row][$column];
    }
    echo '|<br />';
}
```

Both versions of this code produce the same output in the browser:

```
|TIR|Tires|100|OIL|Oil|10|
|SPK|Spark Plugs|4|
```

The only difference between the two examples is that your code will be much shorter if you use the second version with a large array.

You might prefer to create column names instead of numbers, as shown in Figure 3.3. To store the same set of products, with the columns named as they are in Figure 3.3, you would use the following code:

```
$products = array(array('Code' => 'TIR',
                        'Description' => 'Tires',
                        'Price' => 100
                      ),
                  array('Code' => 'OIL',
                        'Description' => 'Oil',
                        'Price' => 10
                      ),
                  array('Code' => 'SPK',
                        'Description' => 'Spark Plugs',
                        'Price' => 4
                      )
                );
```

This array is easier to work with if you want to retrieve a single value. Remembering that the description is stored in the Description column is easier than remembering it is stored in column 1. Using descriptive indices, you do not need to remember that an item is stored at `[x][y]`. You can easily find your data by referring to a location with meaningful row and column names.

You do, however, lose the ability to use a simple `for` loop to step through each column in turn. Here is one way to write code to display this array:

```
for ($row = 0; $row < 3; $row++){
    echo '|'.$products[$row]['Code'].'|'.$products[$row]['Description'].'|';
}
```



```
        '|'.$products[$row]['Price'].'|<br />';  
    }  
}
```

Using a `for` loop, you can step through the outer, numerically indexed `$products` array. Each row in the `$products` array is an array with descriptive indices. Using the `each()` and `list()` functions in a `while` loop, you can step through these inner arrays. Therefore, you can use a `while` loop inside a `for` loop:

```
for ($row = 0; $row < 3; $row++){  
    while (list( $key, $value ) = each( $products[$row])){  
        echo '|'.$value;  
    }  
    echo '|<br />';  
}
```

You do not need to stop at two dimensions. In the same way that array elements can hold new arrays, those new arrays, in turn, can hold more arrays.

A three-dimensional array has height, width, and depth. If you are comfortable thinking of a two-dimensional array as a table with rows and columns, imagine a pile or deck of those tables. Each element is referenced by its layer, row, and column.

If Bob divided his products into categories, you could use a three-dimensional array to store them. Figure 3.4 shows Bob’s products in a three-dimensional array.

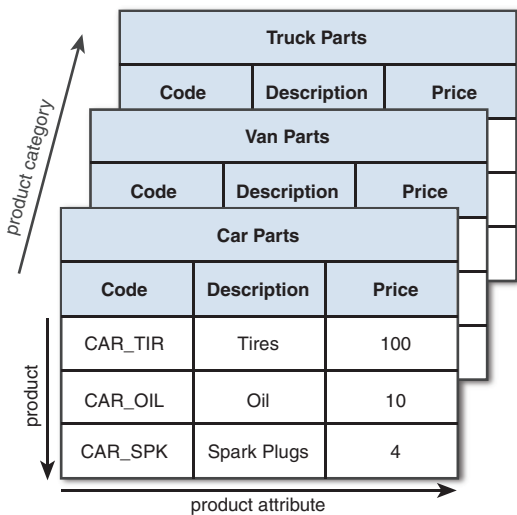


Figure 3.4 This three-dimensional array allows you to divide products into categories

From the code that defines this array, you can see that a three-dimensional array is an array containing arrays of arrays:

```
$categories = array(array(array('CAR_TIR', 'Tires', 100 ),
                             array('CAR_OIL', 'Oil', 10 ),
                             array('CAR_SPK', 'Spark Plugs', 4 )
                          ),
                    array(array('VAN_TIR', 'Tires', 120 ),
                             array('VAN_OIL', 'Oil', 12 ),
                             array('VAN_SPK', 'Spark Plugs', 5 )
                          ),
                    array(array('TRK_TIR', 'Tires', 150 ),
                             array('TRK_OIL', 'Oil', 15 ),
                             array('TRK_SPK', 'Spark Plugs', 6 )
                          )
                    );
```

Because this array has only numeric indices, you can use nested `for` loops to display its contents:

```
for ($layer = 0; $layer < 3; $layer++) {
    echo 'Layer' . $layer . "<br />";
    for ($row = 0; $row < 3; $row++) {
        for ($column = 0; $column < 3; $column++) {
            echo '|' . $categories[$layer][$row][$column];
        }
        echo '|<br />';
    }
}
```

Because of the way multidimensional arrays are created, you could create four-, five-, or even six-dimensional arrays. There is no language limit to the number of dimensions, but it is difficult for people to visualize constructs with more than three dimensions. Most real-world problems match logically with constructs of three or fewer dimensions.

Sorting Arrays

Sorting related data stored in an array is often useful. You can easily take a one-dimensional array and sort it into order.

Using `sort()`

The following code showing the `sort()` function results in the array being sorted into ascending alphabetical order:

```
$products = array('Tires', 'Oil', 'Spark Plugs');
sort($products);
```

The array elements will now appear in the order Oil, Spark Plugs, Tires.

You can sort values by numerical order, too. If you have an array containing the prices of Bob's products, you can sort it into ascending numeric order as follows:

```
$prices = array(100, 10, 4);
sort($prices);
```

The prices will now appear in the order 4, 10, 100.

Note that the `sort()` function is case sensitive. All capital letters come before all lowercase letters. So *A* is less than *Z*, but *Z* is less than *a*.

The function also has an optional second parameter. You may pass one of the constants `SORT_REGULAR` (the default), `SORT_NUMERIC`, `SORT_STRING`, `SORT_LOCALE_STRING`, `SORT_NATURAL`, `SORT_FLAG_CASE`.

The ability to specify the sort type is useful when you are comparing strings that might contain numbers, for example, 2 and 12. Numerically, 2 is less than 12, but as strings '12' is less than '2'.

Passing the `SORT_LOCALE_STRING` constant will sort the array as strings depending on the current locale, as sort orders are different in different locales.

Using `SORT_NATURAL` causes a natural sort order to be used. You can also get this by using the `natsort()` function. Natural sort order is like a combination of string and numeric sorts, to be more intuitive. For example, using a string sort for the strings 'file1', 'file2', and 'file10' would order them as 'file1', 'file10', 'file2'. Using a natural sort, they would be ordered as 'file1', 'file2', 'file10', which is more intuitive—or more natural—for humans.

The constant `SORT_FLAG_CASE` is used in conjunction with `SORT_STRING` or `SORT_NATURAL`. Use the bitwise and operator to combine them, as follows:

```
sort($products, SORT_STRING & SORT_FLAG_CASE);
```

This makes the `sort()` function ignore case, so 'a' and 'A' are treated as equivalent.

Using `asort()` and `ksort()` to Sort Arrays

If you are using an array with descriptive keys to store items and their prices, you need to use different kinds of sort functions to keep keys and values together as they are sorted.

The following code creates an array containing the three products and their associated prices and then sorts the array into ascending price order:

```
$prices = array('Tires'=>100, 'Oil'=>10, 'Spark Plugs'=>4);
asort($prices);
```

The function `asort()` orders the array according to the value of each element. In the array, the values are the prices, and the keys are the textual descriptions. If, instead of sorting by price, you want to sort by description, you can use `ksort()`, which sorts by key rather than value.

The following code results in the keys of the array being ordered alphabetically—Oil, Spark Plugs, Tires:

```
$prices = array('Tires'=>100, 'Oil'=>10, 'Spark Plugs'=>4);
ksort($prices);
```

Sorting in Reverse

The three different sorting functions—`sort()`, `asort()`, and `ksort()`—sort an array into ascending order. Each function has a matching reverse sort function to sort an array into descending order. The reverse versions are called `rsort()`, `arsort()`, and `krsort()`.

You use the reverse sort functions in the same way you use the ascending sort functions. The `rsort()` function sorts a single-dimensional numerically indexed array into descending order. The `arsort()` function sorts a one-dimensional array into descending order using the value of each element. The `krsort()` function sorts a one-dimensional array into descending order using the key of each element.

Sorting Multidimensional Arrays

Sorting arrays with more than one dimension, or by something other than alphabetical or numerical order, is more complicated. PHP knows how to compare two numbers or two text strings, but in a multidimensional array, each element is an array.

There are two approaches to sorting multidimensional arrays: creating a user-defined sort or using the `array_multisort()` function.

Using the `array_multisort()` function

The `array_multisort()` function can be used either to sort multidimensional arrays, or to sort multiple arrays at once.

The following is the definition of a two-dimensional array used earlier. This array stores Bob's three products with a code, a description, and a price for each:

```
$products = array(array('TIR', 'Tires', 100),
                  array('OIL', 'Oil', 10),
                  array('SPK', 'Spark Plugs', 4));
```

If we simply take the function `array_multisort()` and apply it as follows, it will sort the array. But in what order?

```
array_multisort($products);
```

As it turns out, this will sort our `$products` array by the first item in each array, using a regular ascending sort, as follows:

```
'OIL', 'Oil', 10
'SPK', 'Spark Plugs', 4
'TIR', 'Tires', 100
```

This function has the following prototype:

```
bool array_multisort(array &a [, mixed order = SORT_ASC [, mixed sorttype =
SORT_REGULAR [, mixed $... ]]] )
```

For the ordering you can pass `SORT_ASC` or `SORT_DESC` for ascending or descending order, respectively.

For the sort type, `array_multisort()` supports the same constants as the `sort()` function.

One important point to note for `array_multisort()` is that, while it will maintain key-value associations when the keys are strings, it will not do so if the keys are numeric, as in this example.

User-Defined Sorts

Taking the same array as in the previous example, there are at least two useful sort orders. You might want the products sorted into alphabetical order using the description or by numeric order by the price. Either result is possible, but you can use the function `usort()` to tell PHP how to compare the items. To do this, you need to write your own comparison function.

The following code sorts this array into alphabetical order using the second column in the array—the description:

```
function compare($x, $y) {
    if ($x[1] == $y[1]) {
        return 0;
    } else if ($x[1] < $y[1]) {
        return -1;
    } else {
        return 1;
    }
}
usort($products, 'compare');
```

So far in this book, you have called a number of the built-in PHP functions. To sort this array, you need to define a function of your own. We examine writing functions in detail in Chapter 5, “Reusing Code and Writing Functions,” but here is a brief introduction.

You define a function by using the keyword `function`. You need to give the function a name. Names should be meaningful, so you can call it `compare()` for this example. Many functions take parameters or arguments. This `compare()` function takes two: one called `$x` and one called `$y`. The purpose of this function is to take two values and determine their order.

For this example, the `$x` and `$y` parameters are two of the arrays within the main array, each representing one product. To access the `Description` of the array `$x`, you type `$x[1]` because the `Description` is the second element in these arrays, and numbering starts at zero. You use `$x[1]` and `$y[1]` to compare each `Description` from the arrays passed into the function.

When a function ends, it can give a reply to the code that called it. This process is called *returning* a value. To return a value, you use the keyword `return` in the function. For example, the line `return 1;` sends the value 1 back to the code that called the function.

To be used by `usort()`, the `compare()` function must compare `$x` and `$y`. The function must return 0 if `$x` equals `$y`, a negative number if it is less, or a positive number if it is greater. The function will return 0, 1, or -1, depending on the values of `$x` and `$y`.

The final line of code calls the built-in function `usort()` with the array you want sorted (`$products`) and the name of the comparison function (`compare()`).

If you want the array sorted into another order, you can simply write a different comparison function. To sort by price, you need to look at the third column in the array and create this comparison function:

```
function compare($x, $y) {
    if ($x[2] == $y[2]) {
        return 0;
    } else if ($x[2] < $y[2]) {
        return -1;
    } else {
        return 1;
    }
}
```

When `usort($products, 'compare')` is called, the array is placed in ascending order by price.

Note

Should you run these snippets to test them, there will be no output. These snippets are meant to be part of large pieces of code you might write.

The *u* in `usort()` stands for *user* because this function requires a user-defined comparison function. The `uasort()` and `uksort()` versions of `asort` and `ksort` also require user-defined comparison functions.

Similar to `asort()`, `uasort()` should be used when sorting a non-numerically indexed array by value. Use `asort` if your values are simple numbers or text. Define a comparison function and use `uasort()` if your values are more complicated objects such as arrays.

Similar to `ksort()`, `uksort()` should be used when sorting a non-numerically indexed array by key. Use `ksort` if your keys are simple numbers or text. Define a comparison function and use `uksort()` if your keys are more complicated objects such as arrays.

Reverse User Sorts

The functions `sort()`, `asort()`, and `ksort()` all have matching reverse sorts with an *r* in the function name. The user-defined sorts do not have reverse variants, but you can sort a

multidimensional array into reverse order. Because you provide the comparison function, you can write a comparison function that returns the opposite values. To sort into reverse order, the function needs to return 1 if $\$x$ is less than $\$y$ and -1 if $\$x$ is greater than $\$y$. For example,

```
function reverse_compare($x, $y) {
    if ($x[2] == $y[2]) {
        return 0;
    } else if ($x[2] < $y[2]) {
        return 1;
    } else {
        return -1;
    }
}
```

Calling `usort($products, 'reverse_compare')` would now result in the array being placed in descending order by price.

Reordering Arrays

For some applications, you might want to manipulate the order of the array in other ways than a sort. The function `shuffle()` randomly reorders the elements of your array. The function `array_reverse()` gives you a copy of your array with all the elements in reverse order.

Using `shuffle()`

Bob wants to feature a small number of his products on the front page of his site. He has a large number of products but would like three randomly selected items shown on the front page. So that repeat visitors do not get bored, he would like the three chosen products to be different for each visit. He can easily accomplish his goal if all his products are in an array. Listing 3.1 displays three randomly chosen pictures by shuffling the array into a random order and then displaying the first three.

Listing 3.1 **bobs_front_page.php**—Using PHP to Produce a Dynamic Front Page for Bob's Auto Parts

```
<?php
    $pictures = array('brakes.png', 'headlight.png',
                     'spark_plug.png', 'steering_wheel.png',
                     'tire.png', 'wiper_blade.png');

    shuffle($pictures);
?>
<!DOCTYPE html>
<html>
    <head>
        <title>Bob's Auto Parts</title>
```

```
</head>
<body>
  <h1>Bob's Auto Parts</h1>
  <div align="center">
    <table style="width: 100%; border: 0">
      <tr>
        <?php
        for ($i = 0; $i < 3; $i++) {
          echo "<td style='width: 33%; text-align: center'">
            </td>";
          }
        ?>
      </tr>
    </table>
  </div>
</body>
</html>
```

Because the code selects random pictures, it produces a different page nearly every time you load it, as shown in Figure 3.5.

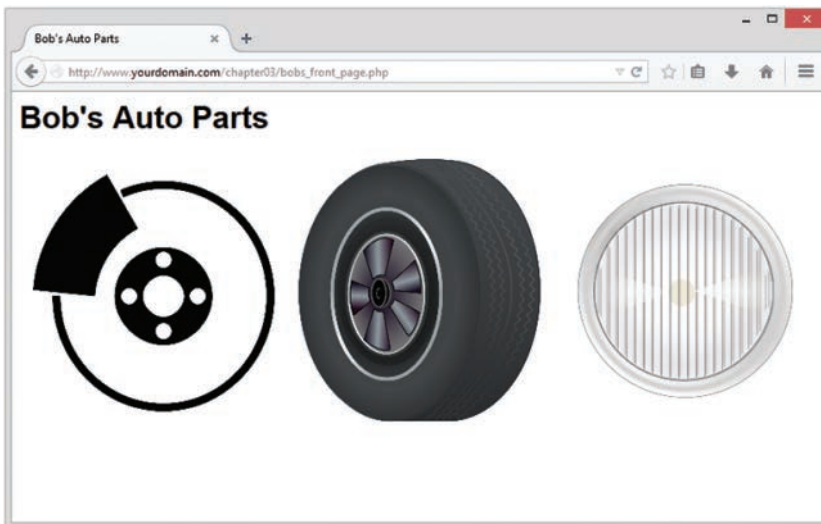


Figure 3.5 The `shuffle()` function enables you to feature three randomly chosen products

Reversing an Array

Sometimes you may want to reverse the order of an array. The simplest way to do this is with the function `array_reverse()`, which takes an array and creates a new one with the same contents in reverse order.

Using the `range()` function usually creates an ascending sequence, as follows:

```
$numbers = range(1,10);
```

You can then use the `array_reverse()` function to reverse the array created by `range()`:

```
$numbers = range(1,10);
$numbers = array_reverse($numbers);
```

Note that `array_reverse()` returns a modified copy of the array. If you do not want the original array, as in this example, you can simply store the new copy over the original.

Alternatively, you could create the array in descending order in the first place, one element at a time, by writing a `for` loop:

```
$numbers = array();
for($i=10; $i>0; $i--) {
    array_push($numbers, $i);
}
```

A `for` loop can go in descending order like this: You set the starting value high and at the end of each loop use the `--` operator to decrease the counter by one.

Here, we create an empty array and then use `array_push()` for each element to add one new element to the end of an array. As a side note, the opposite of `array_push()` is `array_pop()`. This function removes and returns one element from the end of an array.

Note that if the desired array is just a range of descending integers, you can also create it in reverse order by passing `-1` as the optional step parameter to `range()`:

```
$numbers = range(10, 1, -1);
```

Loading Arrays from Files

In Chapter 2, “Storing and Retrieving Data,” you learned how to store customer orders in a file. Each line in the file looked something like this:

```
01:34, 14th September 2015      1 tires 2 oil    3 spark plugs    $145.2    123 Main
Street, Sometown, CA 95128
```

To process or fulfill this order, you could load it back into an array. Listing 3.2 displays the current order file.

Listing 3.2 **vieworders.php**—Using PHP to Display Orders for Bob

```

<?php
    // create short variable name
    $document_root = $_SERVER['DOCUMENT_ROOT'];
?>
<!DOCTYPE html>
<html>
    <head>
        <title>Bob's Auto Parts - Order Results</title>
    </head>
    <body>
        <h1>Bob's Auto Parts</h1>
        <h2>Customer Orders</h2>
        <?php
            $orders= file("$document_root/./orders/orders.txt");

            $number_of_orders = count($orders);
            if ($number_of_orders == 0) {
                echo "<p><strong>No orders pending.<br />
                    Please try again later.</strong></p>";
            }

            for ($i=0; $i<$number_of_orders; $i++) {
                echo $orders[$i]. "<br />";
            }
        ?>
    </body>
</html>

```

This script produces almost exactly the same output as Listing 2.3 in the preceding chapter, which was shown in Figure 2.4. This time, the script uses the function `file()`, which loads the entire file into an array. Each line in the file becomes one element of an array. This code also uses the `count()` function to see how many elements are in an array.

Furthermore, you could load each section of the order lines into separate array elements to process the sections separately or to format them more attractively. Listing 3.3 does exactly that.

Listing 3.3 **vieworders_v2.php**—Using PHP to Separate, Format, and Display Orders for Bob

```

<?php
    // create short variable name
    $document_root = $_SERVER['DOCUMENT_ROOT'];
?>
<!DOCTYPE html>

```

```

<html>
<head>
  <title>Bob's Auto Parts - Customer Orders</title>

  <style type="text/css">
    table, th, td {
      border-collapse: collapse;
      border: 1px solid black;
      padding: 6px;
    }

    th {
      background: #ccccff;
    }
  </style>

</head>
<body>
  <h1>Bob's Auto Parts</h1>
  <h2>Customer Orders</h2>

  <?php
    //Read in the entire file
    //Each order becomes an element in the array
    $orders= file("$document_root/../orders/orders.txt");

    // count the number of orders in the array
    $number_of_orders = count($orders);

    if ($number_of_orders == 0) {
      echo "<p><strong>No orders pending.<br />
        Please try again later.</strong></p>";
    }

    echo "<table>\n";
    echo "<tr>
      <th>Order Date</th>
      <th>Tires</th>
      <th>Oil</th>
      <th>Spark Plugs</th>
      <th>Total</th>
      <th>Address</th>
    <tr>";

    for ($i=0; $i<$number_of_orders; $i++) {
      //split up each line
      $line = explode("\t", $orders[$i]);

```

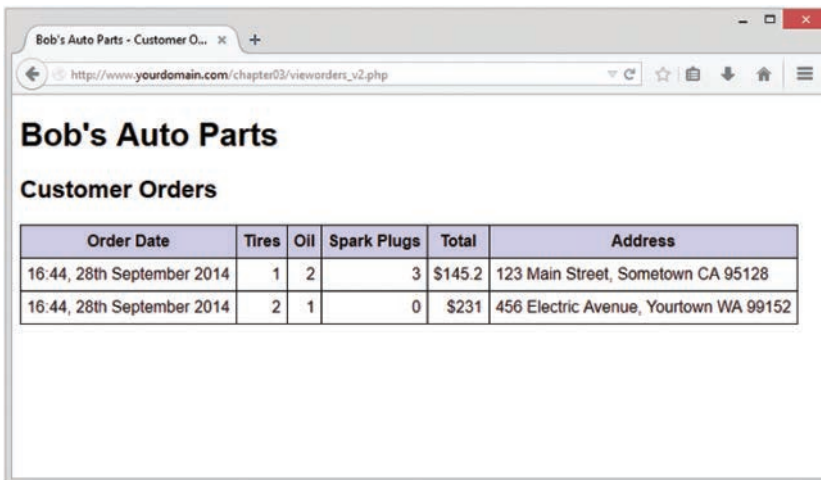
```

// keep only the number of items ordered
$line[1] = intval($line[1]);
$line[2] = intval($line[2]);
$line[3] = intval($line[3]);

// output each order
echo "<tr>
    <td>".$line[0]."</td>
    <td style=\"text-align: right;\">".$line[1]."</td>
    <td style=\"text-align: right;\">".$line[2]."</td>
    <td style=\"text-align: right;\">".$line[3]."</td>
    <td style=\"text-align: right;\">".$line[4]."</td>
    <td>".$line[5]."</td>
</tr>";
}
echo "</table>";
?>
</body>
</html>

```

The code in Listing 3.3 loads the entire file into an array, but unlike the example in Listing 3.2, here we use the function `explode()` to split up each line so that we can apply some processing and formatting before printing. The output from this script is shown in Figure 3.6.



The screenshot shows a web browser window with the title 'Bob's Auto Parts - Customer O...' and the URL 'http://www.yourdomain.com/chapter03/vieworders_v2.php'. The page content includes the heading 'Bob's Auto Parts' and 'Customer Orders'. Below this is a table with two data rows and one header row. The table has six columns: 'Order Date', 'Tires', 'Oil', 'Spark Plugs', 'Total', and 'Address'.

Order Date	Tires	Oil	Spark Plugs	Total	Address
16:44, 28th September 2014	1	2	3	\$145.2	123 Main Street, Sometown CA 95128
16:44, 28th September 2014	2	1	0	\$231	456 Electric Avenue, Yourtown WA 99152

Figure 3.6 After splitting order records with `explode()`, you can put each part of an order in a different table cell for better-looking output

The `explode` function has the following prototype:

```
array explode(string separator, string string [, int limit])
```

In the preceding chapter, we used the tab character as a delimiter when storing this data, so here we do the following:

```
$line = explode("\t", $orders[$i]);
```

This code “explodes” the passed-in string into parts. Each tab character becomes a break between two elements. For example, the string

```
16:44, 28th September 2014\t1 tires\t2 oil\t3 spark plugs\t$145.2\t123 Main Street,  
Sometown CA 95128
```

is exploded into the parts "16:44, 28th September 2014", "1 tires", "2 oil", "3spark plugs", "\$145.2", and "123 Main Street, Sometown CA 95128". Note that the optional *limit* parameter can be used to limit the maximum number of parts returned.

This example doesn’t do very much processing. Rather than output tires, oil, and spark plugs on every line, this example displays only the number of each and gives the table a heading row to show what the numbers represent.

You could extract numbers from these strings in a number of ways. Here, we used the function `intval()`. As mentioned in Chapter 1, `intval()` converts a string to an integer. The conversion is reasonably clever and ignores parts, such as the label in this example, which cannot be converted to an integer. We cover various ways of processing strings in the next chapter.

Performing Other Array Manipulations

So far, we have covered only about half the array processing functions. Many others will be useful from time to time; we describe some of them next.

Navigating Within an Array: `each()`, `current()`, `reset()`, `end()`, `next()`, `pos()`, and `prev()`

We mentioned previously that every array has an internal pointer that points to the current element in the array. You indirectly used this pointer earlier when using the `each()` function, but you can directly use and manipulate this pointer.

If you create a new array, the current pointer is initialized to point to the first element in the array. Calling `current($array_name)` returns the first element.

Calling either `next()` or `each()` advances the pointer forward one element. Calling `each($array_name)` returns the current element before advancing the pointer. The function `next()` behaves slightly differently: Calling `next($array_name)` advances the pointer and then returns the new current element.

You have already seen that `reset()` returns the pointer to the first element in the array. Similarly, calling `end($array_name)` sends the pointer to the end of the array. The first and last elements in the array are returned by `reset()` and `end()`, respectively.

To move through an array in reverse order, you could use `end()` and `prev()`. The `prev()` function is the opposite of `next()`. It moves the current pointer back one and then returns the new current element.

For example, the following code displays an array in reverse order:

```
$value = end ($array);
while ($value){
    echo "$value<br />";
    $value = prev($array);
}
```

For example, you can declare `$array` like this:

```
$array = array(1, 2, 3);
```

In this case, the output would appear in a browser as follows:

```
3
2
1
```

Using `each()`, `current()`, `reset()`, `end()`, `next()`, `pos()`, and `prev()`, you can write your own code to navigate through an array in any order.

Applying Any Function to Each Element

in an Array: `array_walk()`

Sometimes you might want to work with or modify every element in an array in the same way. The function `array_walk()` allows you to do this. The prototype of `array_walk()` is as follows:

```
bool array_walk(array arr, callable func[, mixed userdata])
```

Similar to the way we used `usort()` earlier, `array_walk()` expects you to declare a function of your own. As you can see, `array_walk()` takes three parameters. The first, `arr`, is the array to be processed. The second, `func`, is the name of a user-defined function that will be applied to each element in the array. The third parameter, `userdata`, is optional. If you use it, it will be passed through to your function as a parameter. We'll see how this works shortly.

A handy user-defined function might be one that displays each element with some specified formatting. The following code displays each element on a new line by calling the user-defined function `my_print()` with each element of `$array`:

```
function my_print($value){
    echo "$value<br />";
}
array_walk($array, 'my_print');
```

The function you write needs to have a particular signature. For each element in the array, `array_walk()` takes the key and value stored in the array, and anything you passed as `userdata`, and calls your function like this:

```
yourfunction(value, key, userdata)
```

For most uses, your function will be using only the values in the array. For some, you might also need to pass a parameter to your function using the parameter `userdata`. Occasionally, you might be interested in the key of each element as well as the value. Your function can, as with `my_print()`, choose to ignore the key and `userdata` parameter.

For a slightly more complicated example, you can write a function that modifies the values in the array and requires a parameter. Although you may not be interested in the key, you need to accept it to accept the optional third parameter:

```
function my_multiply(&$value, $key, $factor){
    $value *= $factor;
}
array_walk($array, 'my_multiply', 3);
```

This code defines a function, `my_multiply()`, that will multiply each element in the array by a supplied factor. You need to use the optional third parameter to `array_walk()` to take a parameter to pass to the function and use it as the multiplication factor. Because you need this parameter, you must define the function, `my_multiply()`, to take three parameters: an array element's value (`$value`), an array element's key (`$key`), and the parameter (`$factor`). You can choose to ignore the key.

A subtle point to note is the way `$value` is passed. The ampersand (&) before the variable name in the definition of `my_multiply()` means that `$value` will be *passed by reference*. Passing by reference allows the function to alter the contents of the array.

We address passing by reference in more detail in Chapter 5. If you are not familiar with the term, for now just note that to pass by reference, you place an ampersand before the variable name in the function declaration.

Counting Elements in an Array: `count()`, `sizeof()`, and `array_count_values()`

You used the function `count()` in an earlier example to count the number of elements in an array of orders. The function `sizeof()` is an alias to `count()`. This function returns the number of elements in an array passed to it. You get a count of one for the number of elements in a normal scalar variable and zero if you pass either an empty array or a variable that has not been set.

The `array_count_values()` function is more complex. If you call `array_count_values($array)`, this function counts how many times each *unique* value occurs in the array named `$array`. The function returns an associative array containing a frequency table.

This array contains all the unique values from `$array` as keys. Each key has a numeric value that tells you how many times the corresponding key occurs in `$array`.